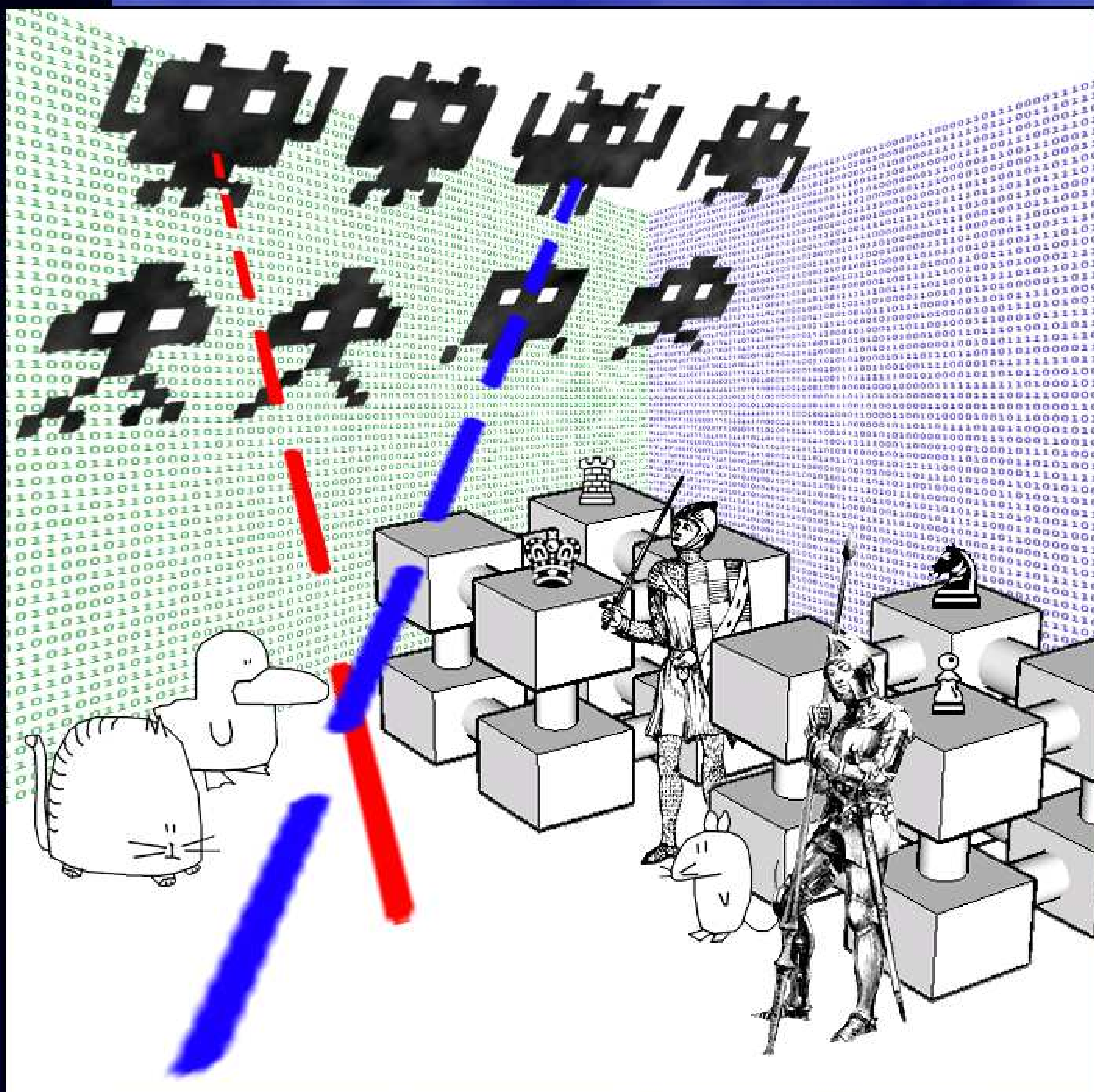
Invent Your Own Computer Games with Python



By Al Sweigart

THE SOMETHING COMPLETELY DIFFERENT SERIES
BOOK ONE

IYOCGwP - Version 4

ISBN 978-0-9821060-0-6

For Caro, with more love
than I ever knew I had.

A Note to Parents and Fellow Programmers

I have more thanks for your interest and more apologies for this book's deficiencies than I can enumerate. My motivation for writing this book comes from a gap I saw in today's literature for kids interested in learning to program. I started programming when I was 9 years old in the BASIC language with a book similar to this one. During the course of writing this, I've realized how a modern language like Python has made programming far easier and more capable. Python has a gentle learning curve while still being a serious language that is used by programmers professionally.

The current crop of programming books for kids that I've seen fell into two categories. First, books that did not teach programming so much as "game creation software" or in dumbed down languages to make programming "easy". Or second, they taught programming like a mathematics textbook: all principles and concepts with application left to the reader. This book takes a different approach: show the game source code right up front and explain programming principles from the examples.

My fellow programmers may notice that the games in this book all use console text, and also use a single stream of text rather than a console window system such as the one the Curses library provides. This is on purpose. Even though there are no graphics or sound, I think that the games are compelling enough in their own right. I also think that graphics and images (and especially game construction kits) mask the true nature of programming. I have the perhaps outdated notion that games do not require fancy graphics to be fun.

The list of things that this book also does not cover: graphics, sound, graphical user interfaces, debugging, file I/O, exceptions, networking, data structures such as stacks and queues, and object oriented programming. After trudging through massively verbose programming manuals myself, I've tried to strip down this book to its most concise form. These other concepts have been reserved for later books.

I have also made this book available under the Creative Commons license, which allows you to make copies and distribute this book (or excerpts) with my full permission, as long as attribution to me is left intact and it is used for noncommercial purposes. I view the last nine months of on and off effort in this book as my gift to world. Thank you again for reading this book.

Al Sweigart
al@coffeeghost.net

The full text of this book is available in HTML or PDF format at:
http://pythonbook.coffeeghost.net

Who is this book for?

- Anyone who wants to teach themselves computer programming, even if they have no previous experience programming.
- Kids and teenagers who want to learn computer programming by programming games. Kids as young as 9 or 10 years old should be able to follow along.
- Adults and teachers who wish to teach others programming.
- Programmers who want to teach others "real" programming by example.

 The Python programming language software this book teaches is also freely available from www.python.org.

Table of Contents

Chapter 1 - "Hello World!", Your First Program

Hello!

This is a book that will show you how to make computer games. All you need is a computer, some software called the Python Interpreter, and this book. The software is free. You can download it at no charge from the Internet. This book will show you how to set up your computer and program a few games. Once you learn how these games work, you will be able to use that knowledge to create games of your own.

When I was a kid, I found a book like this that taught me how to write my first programs and games. It was fun and easy. Now as an adult, I still have fun programming computers as a job, and I get paid for it. But even if you don't become a computer programmer when you grow up, programming is a useful and fun skill to have. (I still sometimes invent my own computer games.)

Computers are very useful machines. In the future, knowing how to program a computer may be as useful as knowing how to read a book. The good news is that learning to program isn't as hard as learning to read. If you can read this book, you can program a computer.

To tell a computer what you want it to do, that is, to program a computer, you will need to learn the computer's language. There are many different programming languages: Basic, Java, Python, Pascal, Haskell, and C++ (pronounced, "see plus plus").

The book I read when I was a kid taught me BASIC programming. Back then, most people who started to learn programming would learn to program in BASIC. But new programming languages have been invented since then. This book is about Python programming. Python is even easier to learn than Basic. Not only is it easy, but it is also a serious and useful programming language. Many adults use Python in their own jobs and hobbies. That's why I chose to make this book about Python programming.

Downloading and Installing Python

You might want the help of an adult or someone else to download and install the Python software. The software that runs programs you write is called an interpreter. The **interpreter** is a program that runs programs written in the Python language. This interpreter program is called the Python interpreter (or sometimes, we just plainly call it Python). You can download the Python interpreter from this website:

http://www.python.org

Click on the Download link on the left side of the web page. On the download page, click on the Python 2.5.1 Windows Installer to download the Python interpreter for Windows. (If you are running an operating system other than Windows, download the Python installer for your operating system instead.) There may be newer versions by the time you read this book. If so, you can download the new version

and the programs in this book will still work.



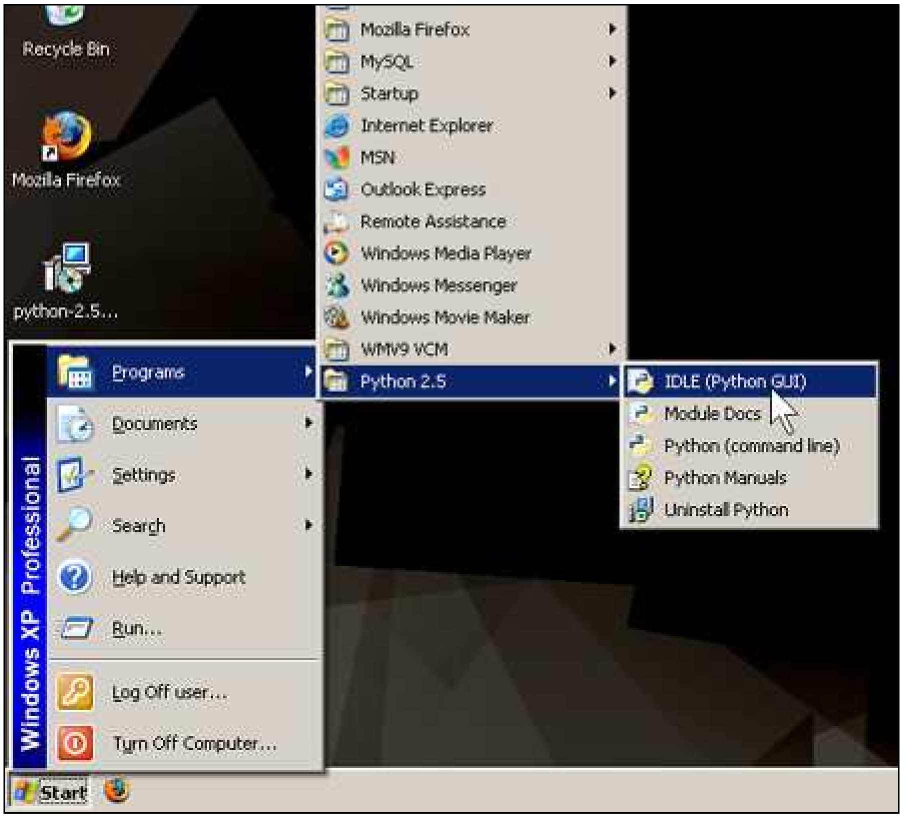
Double-click on the python-2.5.1.msi file that you've just downloaded. This will start the Python installer. All you need to do in the installer is click the Next button. The default choices in the installer are just fine. When the install is finished, click Finish. You may have to restart your computer. You should save any work you have in any other programs that are running, and then click "Yes". Then you will be ready to start programming!

The games we'll create may seem simple compared to games you've played on the XBox, Playstation, or Wii. These games don't have fancy graphics or music. But games don't have to be very complicated to be fun. And unlike those video game consoles, you can always get more games by creating them yourself for free. All you need is a computer and this book.

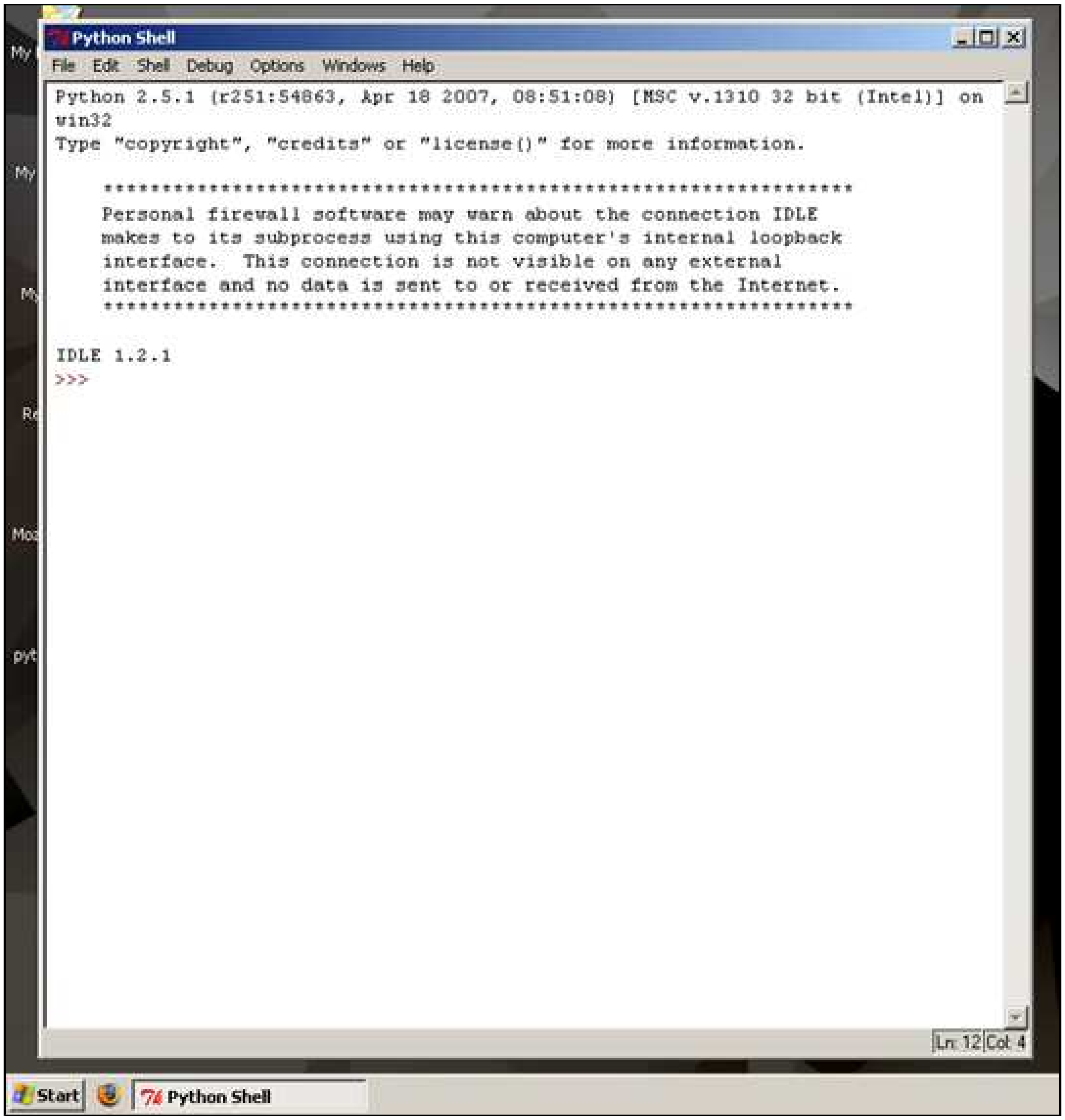
Okay, let's get started!

Starting the Python Interpreter

After you have installed the Python interpreter, you can start it by clicking on Start, then Programs, then Python 2.5, then IDLE (Python GUI). Look at this picture for an example:



You will see a new window with the title, "Python Shell". It will look like this:

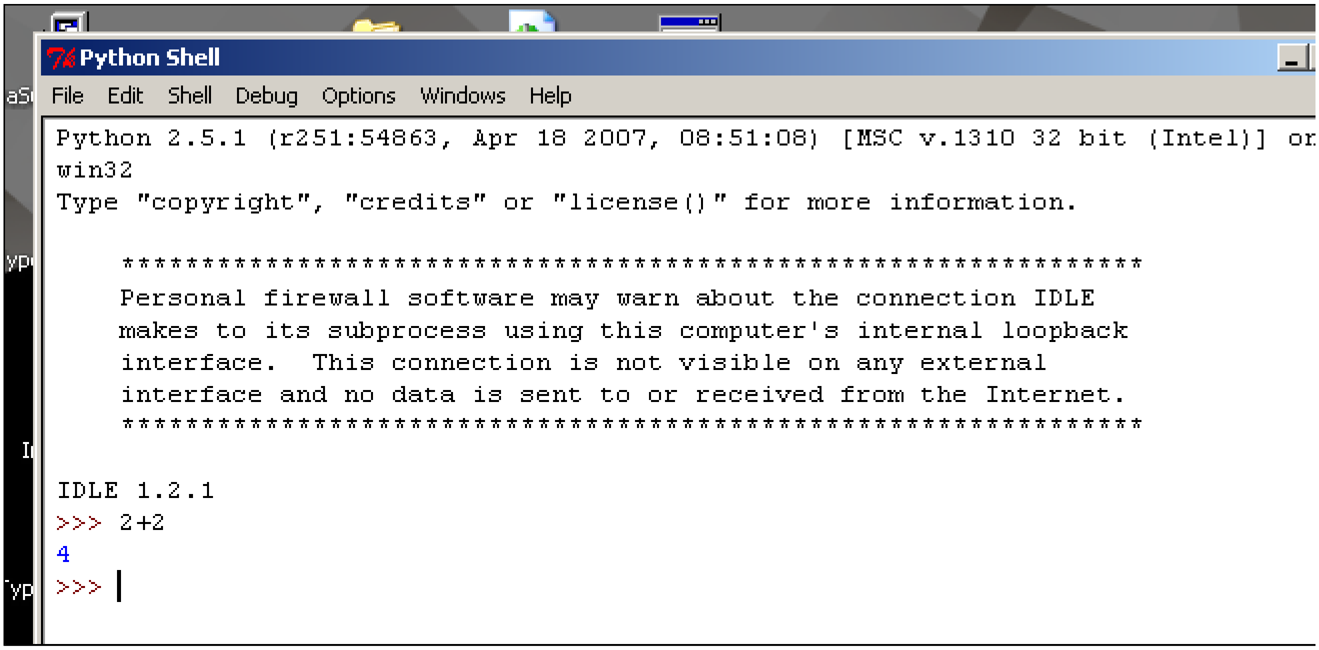


This is the Interactive DeveLopment Environment (IDLE) program. **IDLE** is a program that helps us type in our own programs and games. This window appears when you first run IDLE and is called the **interactive shell**. We can type Python instructions into the shell to make the computer do what we want. A program is a whole bunch of instructions put together, like a story is made up of a whole bunch of sentences.

Let's learn some basic instructions first. We'll learn how to make the computer solve some math problems in the Python shell. Don't worry if you don't know a lot of mathematics. If you know how to add and multiply, you know enough math to do programming. Even if you aren't very good at math, programming is more about problem solving in general than it is about solving math problems.

Some Simple Math Stuff

First, type in `2+2` into the shell and press the Enter key.

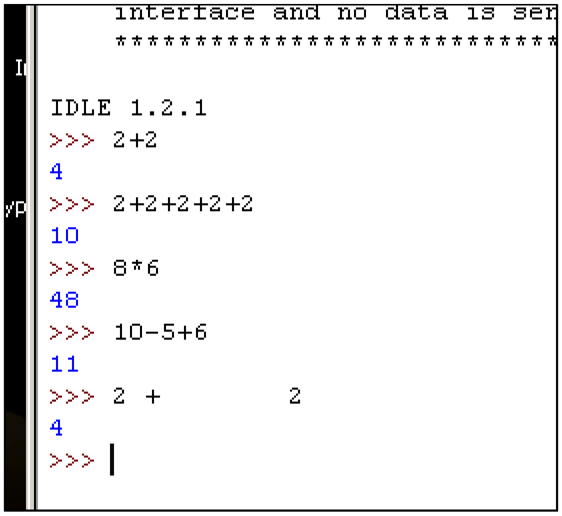


Notice that the Python shell can be used like a calculator. The + sign will do addition and the - sign will do subtraction. The * sign (which is called an **asterisk**) is used for multiplication.

We'll have the computer solve some math problems for us. In programming (and in mathematics), whole numbers are called **integers**. Integers are whole numbers like 4 and 99 and 0. Numbers with fractions or decimal points are not integers. The numbers 3.5 and 42.1 and 5.0 are not integers. In Python, the number 5 is an integer but if we wrote it as 5.0 it would not be an integer. Numbers with the decimal point are called **floating point numbers**.

Try typing some of these math problems into the shell. Remember to press the Enter key after typing each one in.

```
2+2+2+2+2
8*6
10-5+6
2 +        2
```



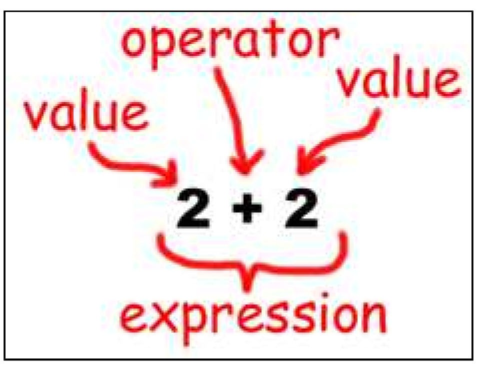
These math problems are called expressions in Python.

You can put any amount of spaces in between the integers and the math signs (which are called **operators**), and Python can tell what you mean. There are other operators besides the mathematical operators, but we will go into them later. These integers are also called **values**. There are other things that are also values (such as strings) which we will talk about later.

Integers are a type of number. Numbers (and integers) are a type of value. Even though integers are numbers, not all numbers are integers. (For example, fractions and numbers with decimal points like `2.5` are numbers that are not integers.) This is like how a cat is a type of pet, but not all pets are cats. Someone could have a pet dog.

Values, operators, and expressions may seem like fancy words for numbers, math signs, and math problems. But knowing these terms will help explain other programming instructions later on.

Actually, expressions include other things besides math problems. An **expression** is made up of values (such as integers like `8` and `6`) connected by an operator (such as the * multiplication sign). A single value by itself is also considered an expression.



The Python shell is handy for solving large math problems very quickly. Try typing in `2063 * 3581`.

```
>>> 2063 * 3581
7387603
>>>
```

Notice that in Python, we don't put commas inside the numbers. We type 2063 instead of 2,063. The computer can do what you tell it to very quickly, but it needs you to tell it in a very specific way. Computer programming is all about writing out precise instructions to get the computer to do exactly what you want.

In the expression `2 + 5 + 7`, the `2 + 5` part is also an expression. Expressions can contain other expressions, like a large Lego building made up of smaller Lego blocks.

So even though computers are very fast and can store a lot of information, they aren't very smart at all. They need human programmers to tell them exactly what to do.

Evaluating Expressions

When the computer solves the expression `10 + 5` and gets the value `15`, we say the computer has evaluated the expression. **Evaluating** an expression reduces the expression to a single value, just like solving a math problem reduces the problem to a single number: the answer. The expressions `10 + 5` and `10 + 3 + 2` have the same value, because they both evaluate to `15`. Remember that single values by themselves are also considered expressions. The expression `15` evaluates to the value `15` (that was pretty easy to evaluate, wasn't it?)

However, if you just type `5 +` into the interactive shell, you will get an error message.

```
>>> 5 +
SyntaxError: invalid syntax
>>>
```

This error happened because `5 +` is not an expression. Expressions have values connected by operators, but `5 +` has an operator that is not connecting two values. This is why the error message appeared. The error message means that the computer does not understand the instruction you gave it.

This may not seem important, but a lot computer programming is about knowing how the computer will evaluate expressions. And remember, expressions are just values connected by operators, or one value by itself.

Variables

When we start programming, we will often want to save the values that our expressions evaluate to so we can use them later. We can store values in things called **variables**. Think of variables like a mailbox that you can put values inside of. You can store values inside variables with the = sign (which is called the **assignment operator**). Try typing `spam = 15` into the shell:

```
>>> spam = 15
>>>
```

This instruction (called an **assignment statement**) stores the value `15` in a variable named `spam`. Unlike expressions, **statements** are instructions that do not evaluate to any value, which is why there is no value that is displayed on the next line in the shell. However, this statement does contain an expression. The value 15 by itself is an expression, which evaluates to the value 15.

Variables store values, not expressions. If we had the statement, `spam = 10 + 5`, then the expression `10 + 5` would first be evaluated down to `15`. Then this `15` value would be the value stored in the variable, `spam`.

You can think of the variable like a mailbox with the value 15 inside of it. The variable name "spam" is the label on the mailbox (so we can tell one mailbox from another) and the value stored in it is like a postcard inside the mailbox.



If we type `spam` into the shell by itself, it will show us what value is stored inside the variable.

```
>>> spam = 15
>>> spam
15
>>>
```

If we type `spam + 5` into the shell, this is the same as `15 + 5` because the value inside `spam` is `15`.

```
>>> spam = 15
>>> spam + 5
20
>>>
```

`spam + 5` is also an expression, just like `15 + 5` would be an expression. When you see a variable

inside an expression, the value that is stored inside the variable is used when the computer evaluate the expression. But we can change which value is stored in the variable by typing in another assignment statement:

```
>>> spam = 15
>>> spam + 5
20
>>> spam = 3
>>> spam + 5
8
>>>
```

Notice that the first time we typed in `spam + 5`, the expression evaluated to `20`. This is because we had stored the value `15` inside the variable spam. But then we stored the value `3` inside of `spam`. The old value of `15` was erased to let the new value of `3` be stored inside the variable. In programming, we say that the value of `15` was **overwritten**. Then, when we typed in `spam + 5`, then that expression evaluates to `8`. If the variable is like a mailbox and the value is like a postcard inside the mailbox, then the mailbox can only hold one postcard at a time.

We can also have expressions on the right side of the = sign. Python will evaluate this expression to get the final value, and then store this value inside of the variable. If you ever want to know what the current value is inside of a variable is, just type the variable name into the shell.

```
>>> spam = 5 + 7
>>> spam
12
>>>
```

Remember, in expressions, the variable acts as a name for a value. We can use the variable as many times as we want. Look at this example:

```
>>> spam = 15
>>> spam + spam
30
>>> spam - spam
0
>>>
```

When the variable spam has the integer value `15` stored in it, then `spam + spam` is the same as `15 + 15`. This is why `spam + spam` evaluates to `30`. And `spam - spam` is the same as `15 - 15`, which evaluates to `0`.

We can even use the value in the `spam` variable to assign `spam` with a new value:

```
>>> spam = 15
>>> spam = spam + 5
>>> spam
20
>>>
```

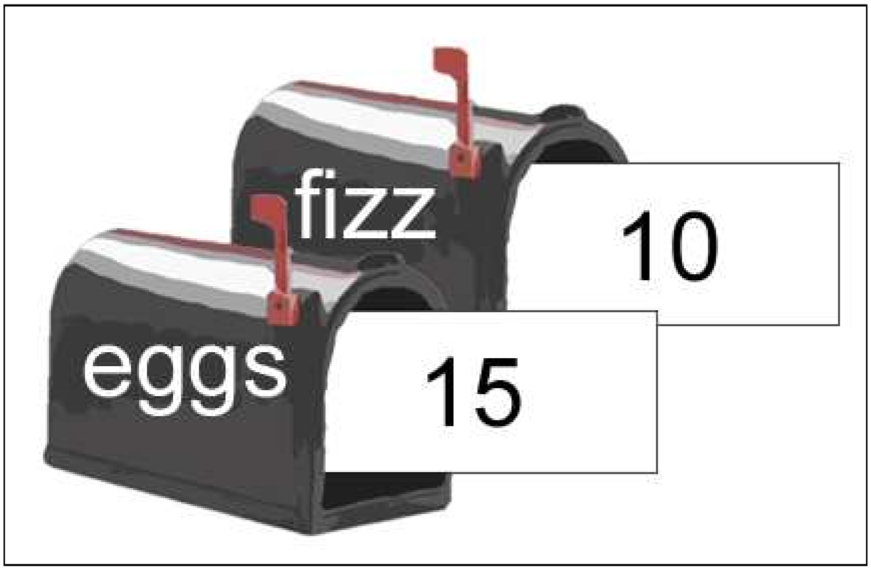
The assignment statement `spam = spam + 5` is sort of like saying "the new value of the spam variable will be the current value of spam plus five." Remember that the variable on the left side of the = sign will be assigned the value that the expression on the right side evaluates to. We can also keep increasing the value in `spam` by `5` several times:

```
>>> spam = 15
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam
30
>>>
```

Let's assign a couple of values to another two variables named `eggs` and `fizz`. We can do this by typing in `fizz = 10`, then press Enter, then type `eggs = 15`, then press Enter.

```
>>> fizz = 10
>>> eggs = 15
>>>
```

These two variables are like two mailboxes, one named `fizz` and the other named `eggs`. The `fizz` variable has `10` inside it, and the `eggs` variable has `15` inside it.



Now let's try assigning a new value to the spam variable. Type `spam = fizz + eggs` into the shell, then press Enter. Then type `spam` into the shell to see what the new value of spam is. Can you guess what it will be?

```
>>> fizz = 10
>>> eggs = 15
>>> spam = fizz + eggs
>>> spam
25
>>>
```

The value in `spam` becomes `25`. This is because when we add `fizz` and `eggs`, we are adding the values stored inside `fizz` and `eggs`.

Strings

That's enough of integers and math for now. Now let's see what Python can do with text. In the Python programming language, we work with little chunks of text called **strings**. We can store string values inside variables just like we can store number values inside variables. When we type strings, we put them in between two single quotes. Try typing `spam = 'hello'` into the shell:

```
>>> spam = 'hello'
>>>
```

The single quotes are not part of the string, they just tell the computer where the string begins and ends. If you type `spam` into the shell to display the contents of spam, it will display the `'hello'` string.

```
>>> spam = 'hello'
>>> spam
'hello'
>>>
```

Strings can have any sort of character or sign in them. Strings can have spaces and numbers as well. These are all strings:

```
'hello'
'Hi there!'
'Albert'
'KITTENS'
'7 apples, 14 oranges, 3 lemons' 'A long time ago in a
galaxy far, far away...'
```

```
'O*&#wY%*&OCfsdYO*&gfC%YO*&%3yc8r2'
```

We can also put string values inside expressions, just like we can put number values inside expressions. The + operator can add one string to the end of another. In programming, we call this **string concatenation**. Try typing `'Hello' + 'World!'` into the shell:

```
>>> 'Hello' + 'World!'
'HelloWorld!'
>>>
```

The string it produces is `'HelloWorld!'`. We should put a space at the end of the `'Hello'` string if we don't want the words bunched together. Try typing `'Hello ' + 'World!'` into the shell:

```
>>> 'Hello' + 'World!'
'HelloWorld!'
>>> 'Hello ' + 'World!'
'Hello World!'
>>>
```

You can't add a string to an integer, or an integer number to a string. This is because a string and an integer are different **data types**. The data type of the value `'Hello'` is a string. The data type of the value `5` is an integer.

Adding `5` and the string `'Hello'` doesn't really make any sense anyway. If we tried to do it, Python would think we were trying to concatenate a string and an integer, or maybe trying to add an integer and a string, and give us an error:

```
>>> 'Hello ' + 5

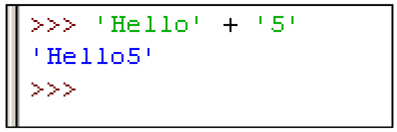
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    'Hello ' + 5
TypeError: cannot concatenate 'str' and 'int' objects
>>> 5 + 'Hello'

Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    5 + 'Hello'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

These error messages may look strange and confusing, but later we'll learn what they mean and how they can help us figure out what went wrong.

However, there is a difference between using the integer `5` and the string `'5'`. You can tell that `'5'`

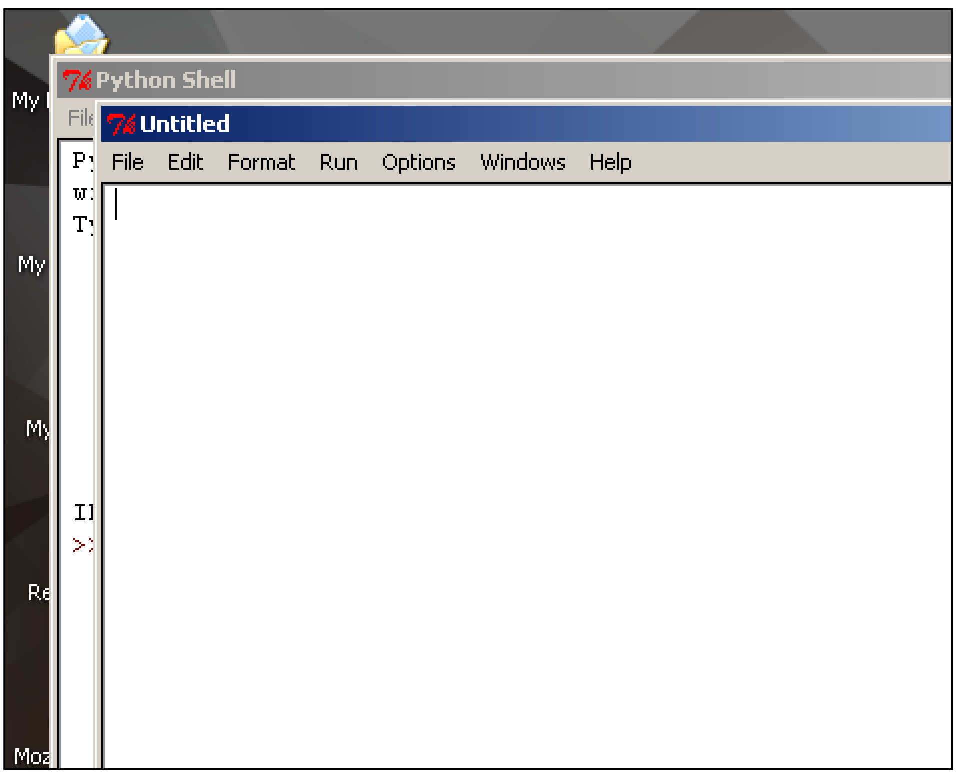
is a string because it has quotes around it.



You may have noticed that the IDLE program makes strings appear in green text to help make them stand out while you type them. The value that the expression evaluates to, however, will show up in blue in the shell no matter what the data type.

Writing Programs

Let's write our first program! Until now we have been typing instructions one at a time into the interactive shell. When we write programs though, we type in several instructions and have them run all at once. Click on the File menu at the top of the Python Shell window, and select New Window. A new blank window will appear. We will type our programs into this window, which is called the **file editor**.

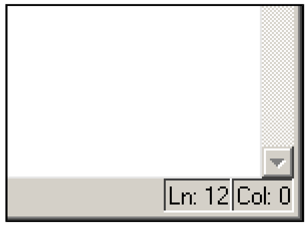


Hello World!

A tradition for programmers learning a new language is to make their first program display the text "Hello world!" on the screen. We'll create our own Hello World program now.

You don't have to type in the numbers or period on the left side of the source code. That's just there so we can refer to each line by number in our explanation. If you look at the bottom-right corner of the

source code window, it will tell you which line the cursor is currently on.



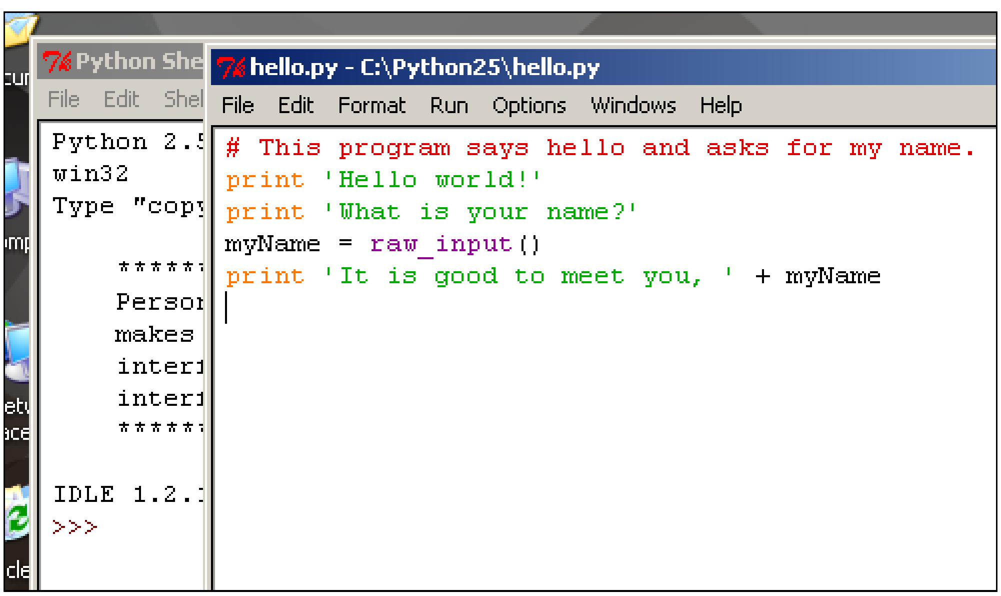
In the picture, the cursor is currently on line 12.

Type the following text into this new window. We call this text the **source code** of the program. These are the instructions to the Python interpreter that explain exactly how the program should behave. (Do not type the numbers at the beginning of each line. Those numbers are for making this book more readable, and they are not part of the source code.)

hello.py

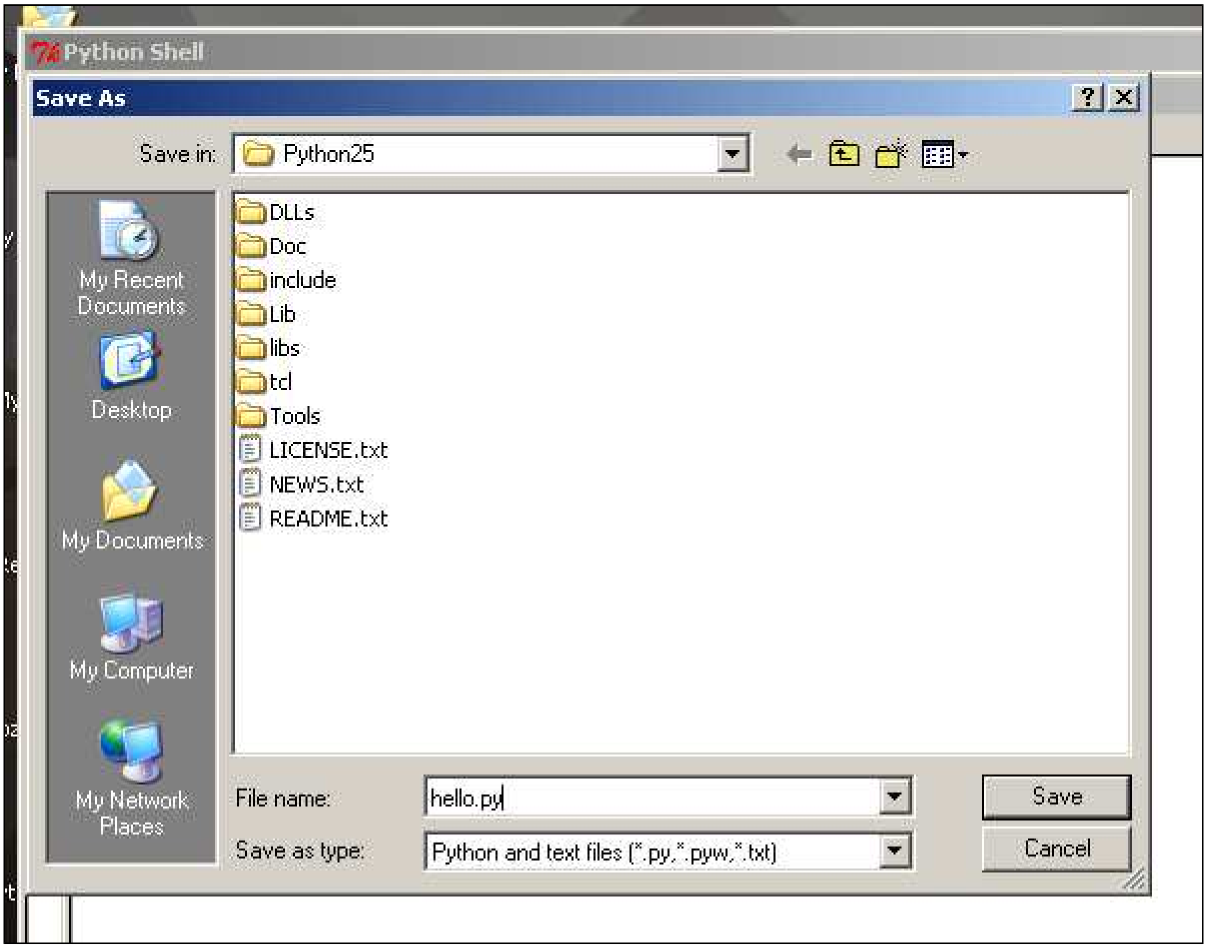
```
# This program says hello and asks for my name.
print 'Hello world!'
print 'What is your name?'
myName = raw_input()
print 'It is good to meet you, ' + myName
```

The IDLE program will give different types of instructions different colors. After you are done typing this code in, the window should look like this:



We will want to save this source code so we don't have to retype it each time we start IDLE. Click on

the File menu at the top, and then click on Save As. A new window will open that asks us what name we want to give this file. Type in `hello.py`, so it looks like this:

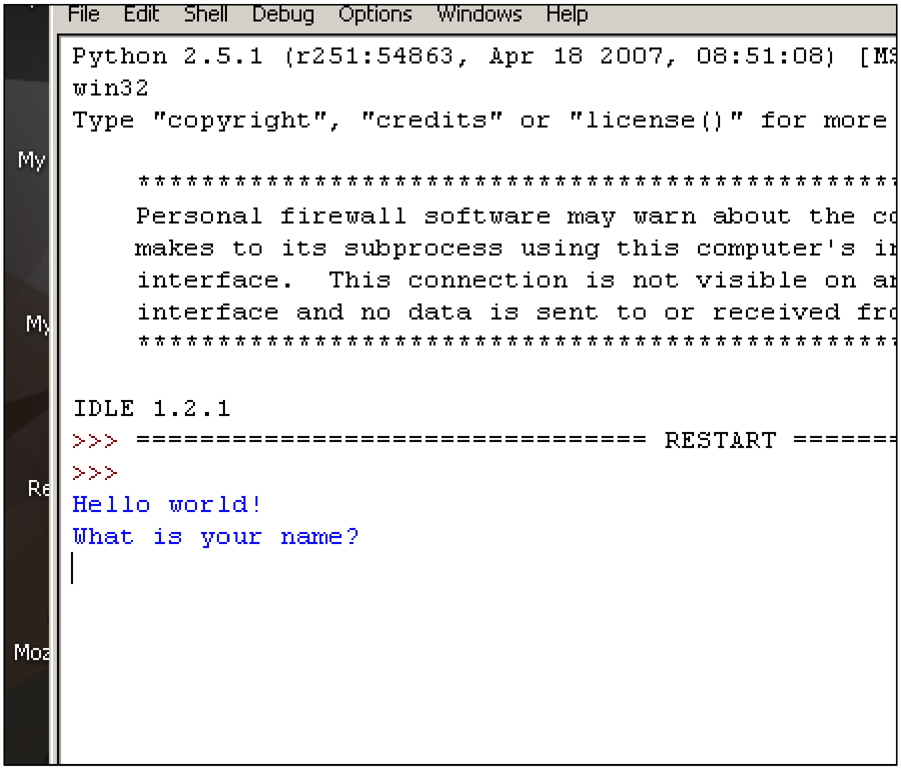


Then click on the Save button.

You should save your program every once in a while as you type them. If the computer crashes or you accidentally exit from IDLE, any typing you have done since you last saved will be lost.

To load this saved program later, click on the File menu at the top, and then click on Open. A new window will appear that asks you to choose which file to open. Click on hello.py and then click on the Open button.

Now we want to run the program we have just typed in. Click on the Run menu at the top, and then click on Run Module. Or, instead of clicking on the menu, you can just push the F5 key on your keyboard. The program will run in the Python Shell window that appeared when we first ran the IDLE program. Our program asks us for our name. Go ahead and type it in, and then press the Enter key.



When we push Enter, the program will greet the user by name. We call the person who will run and use the program the **user**. We call the person who wrote the program the **programmer**. Congratulations! You've written your first program. You are now a computer programmer. You can run this program again if you like. Just click on the window with our source code again, and click on the Run menu, then Run Module. (Or press F5. This is what I like to do since it is quicker.)

How does this program work? Well, each line that we typed in is performed one after the other. The program starts at the very top and then **executes** each line. After the program executes the first line, it moves on and executes the second line, and then it executes the third line, and so on.

Think of the program like a cake recipe. The recipe tells you the exact steps you need to take to bake a cake. Do the first step first, then the second, and keep going until you reach the end. The instructions in your program are executed one by one starting from the top and then going down. We call this the **flow of execution**, or just the execution for short.

Code Explanation

So what does all of that code we typed in mean? Let's look at each line we typed in, one line at a time.

```
# This program says hello and asks for my name.
```

This line is called a **comment**. Comments are ignored in the program. Comments are not for the computer, but for the programmer. They are there to remind the programmer of what the program does. Any text after the # sign (called the **pound sign**) is a comment. Programmers usually put comments at the top of their code to give their program a title. The IDLE program (the program that we are typing our code into) makes comments appear in red text to help make them stand out.

```
2. print 'Hello world!'
```

This line is a **print statement**. A `print` statement is the `print` keyword followed by an expression. The statement will display the evaluated expression on the screen. Unlike typing strings into the shell, when we write a program, the value that an expression evaluates to is not displayed on the screen. To display the expression's value on the screen, we use a `print` statement. We want to display `Hello world!` on the screen, so we type the print keyword followed by the 'Hello world!' string.

The Difference Between Statements and Expressions

What is the difference between a statement and an expression? All expressions evaluate to a single value, and statements do not evaluate to anything. An expression is made up of values connected by operators, which evaluate to a single value (for example, `2 + 3` evaluates to `5`. But the `print` statement does not evaluate to a value.

You could not assign the value that a `print` statement evaluates because statements do not evaluate to values. If you tried, you would get a syntax error. (Just for fun, try typing it into the interactive shell.) A **syntax error** happens when Python cannot understand what you're program is trying to do.

```
>>> spam = print 'Hello!'
SyntaxError: invalid syntax
>>>
```

In fact, you can type a `print` statement into the shell:

```
>>> 'hello'
'hello'
>>> print 'hello'
hello
>>>
```

When the `print` statement runs, it shows the string itself without the quotes. But remember that in programs, nothing will appear on the screen unless you use a `print` statement.

Code Explanation continued...

```
3. print 'What is your name?'
```

This line is also a `print` statement. This time, the program will display `What is your name?`.

```
4. myName = raw_input()
```

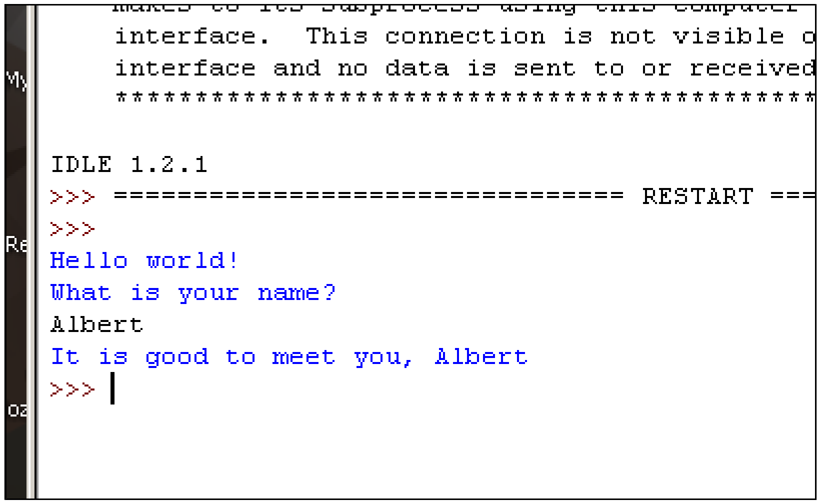
This line has a variable and a **function call**. The variable is named `myName` and the function is named `raw_input()`. A **function** is a bit of code that does a particular action. When we call a function, the program does whatever the function is programmed to do. When `raw_input()` is called, the program waits for the user to type in text and press Enter. This text string is what the function call to the `raw_input()` evaluates to. The value that a function call will evaluate to is called the **return value**. The `raw_input()` function returns the string that the user typed in. Because function calls can be evaluated, they can also be part of an expression. Then this looks like a regular assignment where `myName` stores a string inside it.

Notice that when I talk about the `raw_input()` function, I add parentheses to the end of it. This is how we type out function names, because if I just wrote `raw_input` you would not know if I meant a variable named `raw_input` or a function named `raw_input`. The parentheses at the end let us know we are talking about a function, much like the quotes in `'42'` let us know we are talking about the string `'42'` and not the integer `42`.

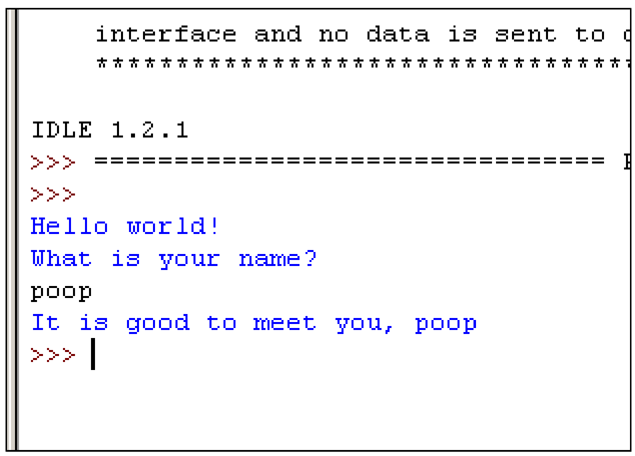
```
5. print 'It is good to meet you, ' + myName
```

On the last line we have a `print` statement again. This time, we use the plus operator (+) to concatenate the string `'It is good to meet you, '` and the string stored in the `myName` variable. This is how we get the program to greet us by name.

After the program executes the last line, it stops. Programmers say the program has **terminated** or **exited**. All of the variables are forgotten by the computer, including the string we stored in `myName`. Try running the program again and enter a different name.



Remember, the computer only does exactly what you program it to. In this program, it is programmed to ask you for your name, let you type in a string, and then it will say hello and display the string you typed. But you don't have to type in your name. You can type in anything you want and the computer will treat it the same:



"My Favorite Stuff"

Let's make another program. Open a new window by clicking on the File menu at the top and then clicking on New Window. (And remember to not type the numbers at the beginning of each line. Those are only to make the source code more readable here.)

favorites.py

```
# Favorite stuff
print 'Tell me what your favorite color is.'
favoriteColor = raw_input()

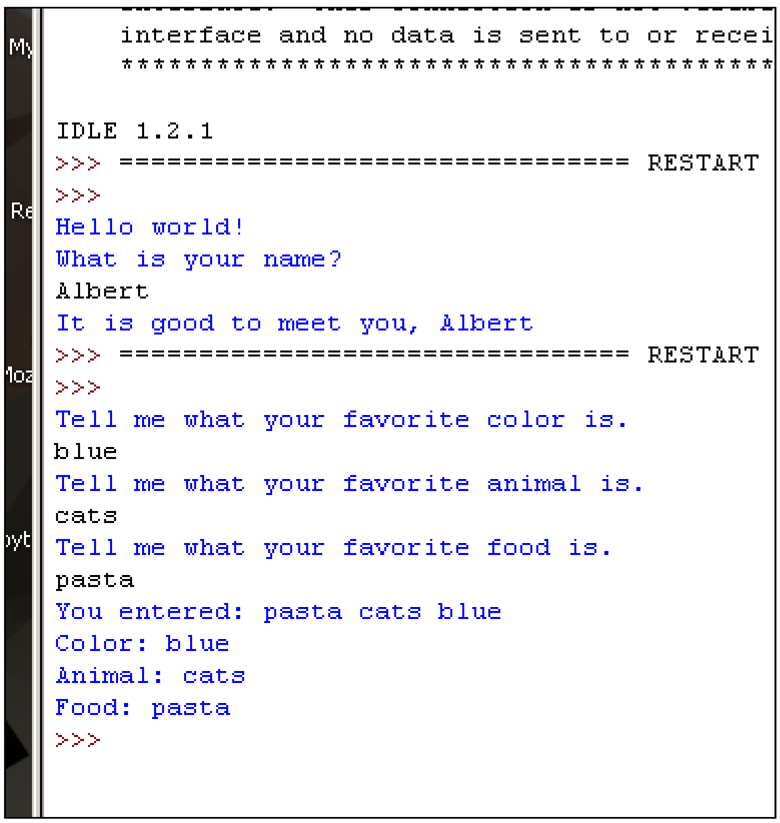
print 'Tell me what your favorite animal is.'
```

```
favoriteAnimal = raw_input()

print 'Tell me what your favorite food is.'
favoriteFood = raw_input()

# display our favorite stuff
print 'You entered: ' + favoriteFood + ' ' +
favoriteAnimal + ' ' + favoriteColor
# print 'Here is a list of your favorite things.'
print 'Color: ' + favoriteColor
print 'Animal: ' + favoriteAnimal
print 'Food: ' + favoriteFood
```

Save this program as `favorites.py` and then press F5 to run it.



Code Explanation

This program looks similar to our Hello World program. Let's look at each line carefully.

```
# Favorite stuff
```

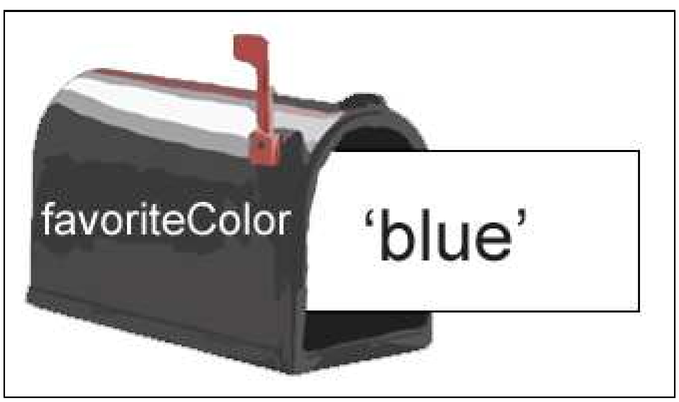
This is another comment. The program will ignore it. It's just there to remind us what this program does if we look at the source code later.

```
print 'Tell me what your favorite color is.'
```

Here we display a bit of text asking the user to type in their favorite color by using the `print` keyword.

```
favoriteColor = raw_input()
```

Now we are going to call the `raw_input()` function to let the user type in their favorite color. When they press enter, the string the user entered is stored in the `favoriteColor` variable.




```
print 'Tell me what your favorite animal is.'
favoriteAnimal = raw_input()
```

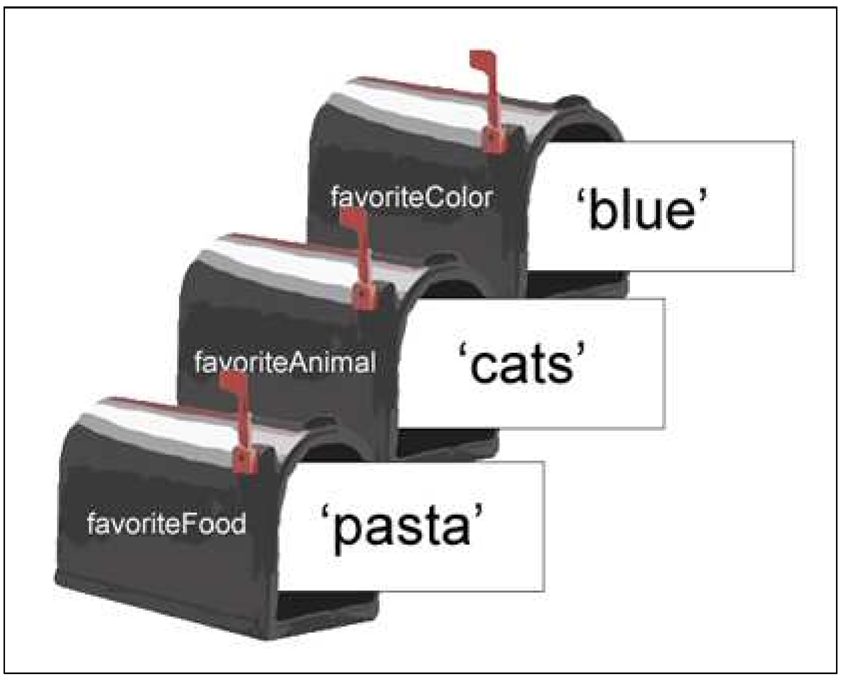
These two lines are similar to the ones before. Notice that there is a blank line in between them. In the

Python language, blank lines are just ignored. This is helpful because then we don't have to have all the lines bunched together.

This time, the user will type what their favorite animal is, and the string will be stored in a variable named `favoriteAnimal`.

```
print 'Tell me what your favorite food is.'
favoriteFood = raw_input()
```

Finally, we will let the user type in their favorite food. This string is stored in yet another variable called `favoriteFood`.




```
# display our favorite stuff
```

Here's another comment. Comments don't always have to go at the top of the program. They can show up anywhere. All the text after the pound sign (#) will be ignored by the program and won't be shown to the user. It just reminds the programmer what the program does.

```
print 'You entered: ' + favoriteFood + ' ' +
favoriteAnimal + ' ' + favoriteColor
```

This `print` statement will show us the favorite food, animal, and color we entered. The plus sign is used to combine the string `'You entered: '` with the strings we stored in our variables earlier. We don't want the strings in the variable to be bunched together, so we add a string with one space in between them. This will make the entire string look something like this:

```
'You entered: pasta cats blue'
```

Instead of this:

```
'You entered: pastacatsblue'
```

```
# print 'Here is a list of your favorite things.'
```

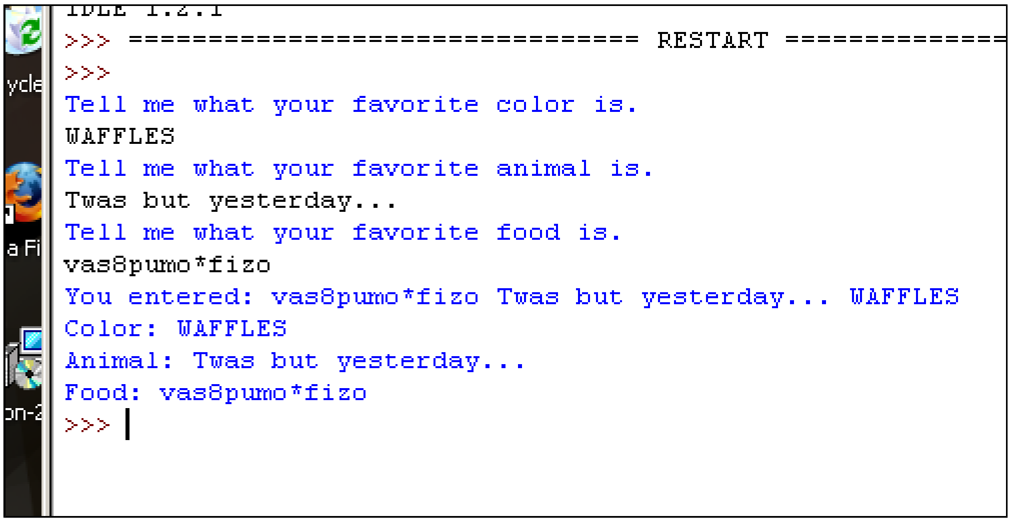
This line looks like another `print` statement. But do you see the pound sign at the start of it? That means this line is really a comment and the program ignores this code. Sometimes the programmer may want to remove code from the source code with the intent to add it back in later. Instead of deleting the code, you can just put a pound sign to have it ignored for now. If you delete the pound sign, then this code will no longer be a comment and would be executed with the rest of the program. In IDLE, you can easily see that this is a comment and not code because it is in red text.

```
print 'Color: ' + favoriteColor
print 'Animal: ' + favoriteAnimal
print 'Food: ' + favoriteFood
```

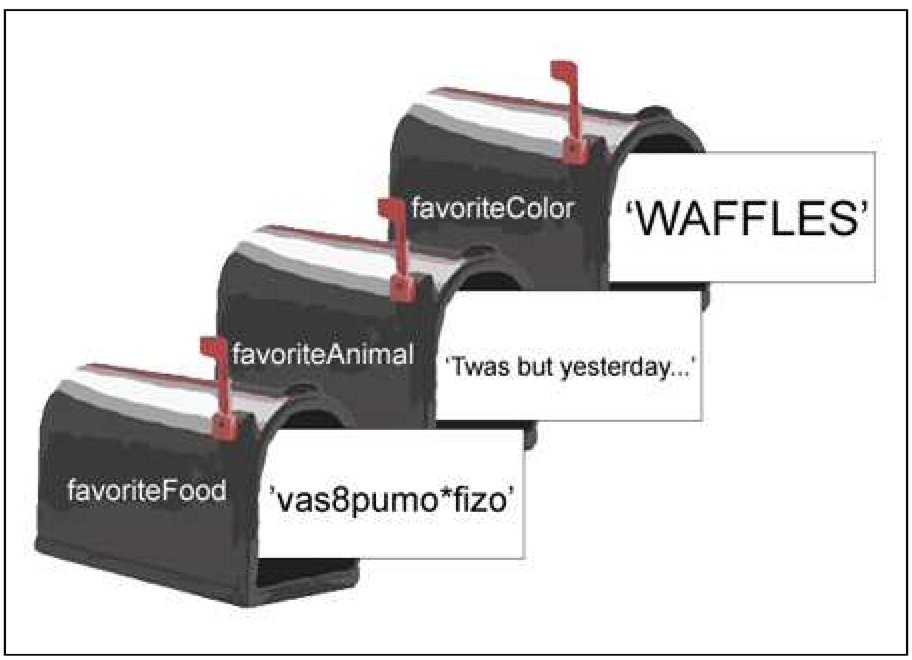
These three lines will display our favorite things once again. When the last line of the program executes, the program terminates.

Crazy Answers and Crazy Names for our Favorite Stuff

The computer doesn't really care what you type in. It doesn't understand what food or animals or colors are. All it knows is that the user will type in some string. We don't have to type in our favorite things at all. Look at this run of the program where I type in some crazy answers:



All the program understands is that it should store the string the user enters into the variables and display the string in those variables later on.



The program also does not care what name we give to our variables. Our program would work just the same if it looked like this:

favorites2.py

```
# Favorite stuff 2
print 'Tell me what your favorite color is.'
q = raw_input()

print 'Tell me what your favorite animal is.'
fizzy = raw_input()

print 'Tell me what your favorite food is.'
AbrahamLincoln = raw_input()

# display our favorite stuff
print 'You entered: ' + q + ' ' + fizzy + ' ' + AbrahamLincoln
#print 'Here is a list of your favorite things.'
print 'Color: ' + q
print 'Animal: ' + fizzy
print 'Food: ' + AbrahamLincoln
```

The names we give the variables are more for our benefit than the computer's benefit. One name looks the same as any other to the computer. The name `q` doesn't help us remember that this variable is supposed to store the string of the user's favorite color. And the name `fizzy` isn't any type of animal. And using the name `AbrahamLincoln` for the variable to store our favorite color is just silly. But since we use the variables in the same way as before, the program works the exact same.

Capitalizing our Variables

Have you noticed that variable names that are made up of more than one word have the other words capitalized? This is to make the variable names easier to read because variable names can't have spaces in them.

```
thisnameiskindofhardtoread
thisNameIsEasierToRead
```

Leave the first word in lowercase, but start the other words in uppercase. We call something in a certain way like this a **convention**: we don't have to do it this way, but doing it this way makes it a little easier. The convention for capitalizing variable names is to leave the first word in lowercase but start the other words in uppercase.

Remember, the computer doesn't care how we name our variables. It only cares how we use them in the program. Look at this program:

favorites3.py

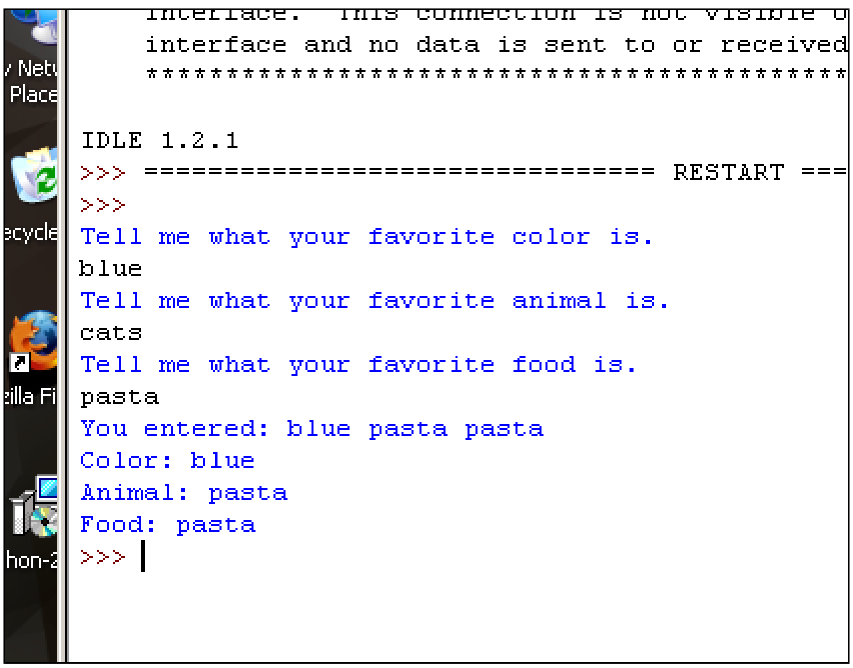
```
# Favorite stuff 3
print 'Tell me what your favorite color is.'
q = raw_input()

print 'Tell me what your favorite animal is.'
AbrahamLincoln = raw_input()

print 'Tell me what your favorite food is.'
AbrahamLincoln = raw_input()

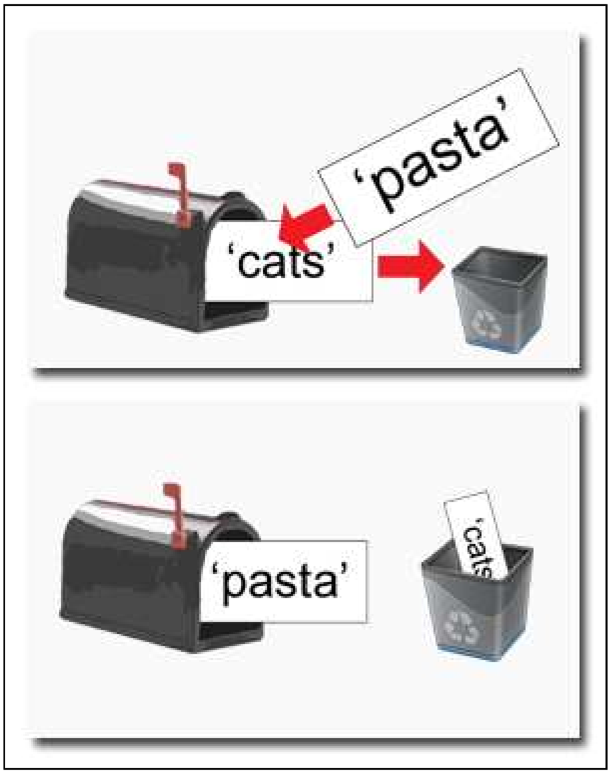
# display our favorite stuff
print 'You entered: ' + q + ' ' + AbrahamLincoln + ' '
    + AbrahamLincoln
#print 'Here is a list of your favorite things.'
print 'Color: ' + q
print 'Animal: ' + AbrahamLincoln
print 'Food: ' + AbrahamLincoln
```

When we run this program, it looks like this:



What happened here? The favorite animal and favorite food are the same thing. If you notice, we use the same variable named `AbrahamLincoln` to store a string of our favorite animal and our favorite food. When the user typed in their favorite animal, this string was stored in the `AbrahamLincoln` variable. But when the user typed in their favorite food, this string was also stored in the `AbrahamLincoln` variable and the favorite food string was forgotten. The favorite food value was overwritten. The computer can't tell the difference between them because they use the same name. So the

computer thinks we mean to use the same variable.

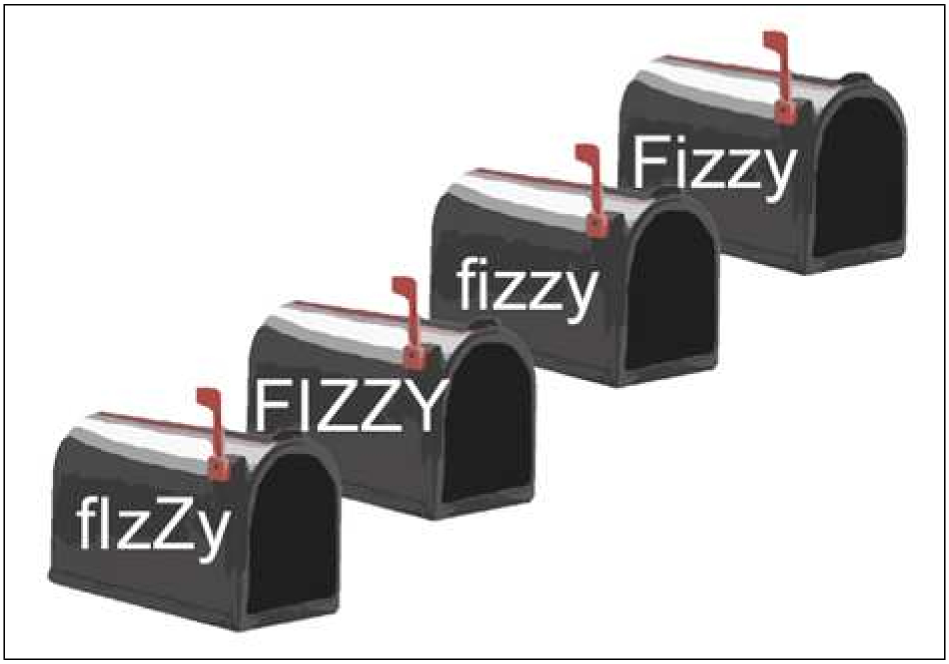


A variable can only store one value at a time.

The computer will do exactly what we tell it to do, even if we tell it to do the wrong thing. The computer can't read our minds and figure out what we want it to do. It is up to the programmer to make sure the program works just right.

As a final note about variable and function names, I should tell you that the computer does pay attention to the capitalization of the name. The computer considers these names to be four separate variables:

```
fizzy
Fizzy
FIZZY
fIzZy
```



Four differently-cased names means four different variables.

We call this **case-sensitivity**. In the Python language, variable and function names are case-sensitive. If you try to call the `RAW_INPUT()` function instead of the `raw_input()` function, you will get an error because the computer doesn't know of a function named `RAW_INPUT()`. It only knows a function named `raw_input()`.

So remember that even though the computer doesn't care what you name your variables or how you capitalize them, be sure to always use the same capitalization. It is also a convention to never use two different variables with the same name but different capitalization. If you use the variable `favoriteFOOD` to store the string of your favorite breakfast food and the variable `FAVORITEfood` to store your favorite dinner food, it is easy to forget which is which.

You don't always have to finish typing in a program before you run it. You can just have some of the code complete, and then run it just to see how the program behaves. Programmers will often type some code, run the program, type some more code, run the program again, and so on in order to make sure the code is coming along the way they like. You can also always use the interactive shell to type single lines of code in to see what it does.

Now that we have some of the basics down, in the next chapter we will create our first game!

Things Covered In This Chapter:

- Downloading and installing the Python interpreter.
- Using IDLE's interactive shell to run instructions.
- Flow of execution
- Expressions, and evaluating expressions

- Integers
- Operators (such as + - *)
- Variables
- Assignment statements
- Overwriting values in variables.
- Strings
- String concatenation
- Data types (such as strings or integers)
- Using IDLE to write source code.
- Saving and running programs in IDLE.
- The `print` statement.
- The `raw_input()` function.
- Comments
- Case-sensitivity
- Conventions

Chapter 2 - Guess the Number

We are going to make a "Guess the Number" game. In this game, the computer will think of a random number from 1 to 20, and ask you to guess the number. You only get six guesses, but the computer will tell you if your guess is too high or too low. If you guess the number within six tries, you win.

Because this program is a game, we'll call the user the **player**.

First, type this code in exactly as it appears here, and then save it by clicking on the File menu and then Save As. Give it a file name like, `guess.py`. Then run it by pressing the F5 key. Don't worry if you don't understand the code now, I'll explain it step by step.

Be sure to type it exactly as it appears. Some of the lines don't begin at the leftmost edge of the line, but are indented by four or eight spaces. Be sure to put in the correct amount of spaces for the start of each line.

Some of these lines are too long to fit on one line in the page, and it wraps around to the next line. When you type them into the file editor, type these lines of code all on the same line. You can tell if a new line starts or not in this book by the line numbers on the left side. For example, this has only two lines of code, even though the first line wraps around:

```
1. print 'This is the first line! xxxxxxxxxx
   xxxxxxx'
2. print 'This is the second line!'
```

Sample Run

Here is the text from a sample run of this game. The text that the program prints out is in blue, and the text that the player types in is in black and in bold.

```
Hello! What is your name?
Albert
Well, Albert, I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too high.
Take a guess.
2
Your guess is too low.
Take a guess.
4
Good job, Albert! You guessed my number in 3 guesses!
```

Source Code

guess.py

```
# This is a guess the number game.
import random

guessesTaken = 0

print 'Hello! What is your name?'
myName = raw_input()

number = random.randint(1, 20)
print 'Well, ' + myName + ', I am thinking of a number between 1 and 20.'

while guessesTaken < 6:
    print 'Take a guess.' # There are four spaces in front of print.
    guess = raw_input()
    guess = int(guess)

    guessesTaken = guessesTaken + 1

    if guess < number:
        print 'Your guess is too low.' # There are eight spaces in front of print.

    if guess > number:
        print 'Your guess is too high.'

    if guess == number:
        break

if guess == number:
    guessesTaken = str(guessesTaken)
    print 'Good job, ' + myName + '! You guessed my number in ' + guessesTaken + ' guesses!'

if guess != number:
    number = str(number)
    print 'Nope. The number I was thinking of was ' + number
```

Even though we are typing in our source code into this file editor new window, we can still go back to the shell to type in individual instructions to see what they do. The interactive shell is very good for

experimenting with different instructions when we are not running a program.

Code Explanation

Let's look at each line of code.

```
1. # This is a guess the number game.
```

This is a comment. Remember that Python will ignore everything after the # sign. This just reminds us what this program does.

```
2. import random
```

This is an **import statement**. The `import` statement is not a function (it does not have parentheses after its name). The statement has a special Python keyword, like the `print` statement has, called the `import` keyword. Many functions like `raw_input()` are included with every Python program. But some functions exist in separate programs called modules. **Modules** are other Python programs that contain other functions that we can use. The `import` statement will add these modules and their functions to our program.

The `import` statement is made up of the `import` keyword followed by the module name.

This line imports a module named `random`. The `random` module has several functions related to random numbers. We'll use one of these functions later to have the computer come up with a random number for us to guess.

```
4. guessesTaken = 0
```

This creates a new variable named `guessesTaken`. We will store the number of guesses we've made in this variable. Since the player hasn't made any guesses so far, we will store the integer `0` here.

```
print 'Hello! What is your name?'
myName = raw_input()
```

These two lines are identical to our Hello World program. Programmers will often reuse code from heir other programs when they need the program to do something similar. When these two lines are done executing, the string of the player's name will be stored in the `myName` variable. (Remember, the string night not really be the player's name. It's just whatever string the player typed in.)

```
number = random.randint(1, 20)
```

Here we are calling a new function named `randint()`, and then storing the return value in a variable named `number`. Because `randint()` is one of the functions that the `random` module provides, we put `random.` (that is, the word "random" followed by a period) in front of it to tell our program the function s in the `random` module. The `randint()` function will return a random integer between (and ncluding) the two integers we give it. Here, we give it the integers 1 and 20 between the parentheses that follow the function name (separated by a comma). Whatever the random integer that `randint` has returned is, it is stored in a variable named `number`.

Just for a moment, go back to the interactive shell and type `import random` to import the random nodule. Then type `random.randint(1, 20)` to see what the function call evaluates to. It will return an integer that between 1 and 20. Type it again, and the function call will probably evaluate to a different nteger. This is because each time the `randint()` function is called, it will evaluate to some random number. This is like when you roll some dice, you will come up with a random number each time.

```
>>> import random
>>> random.randint(1, 20)
12
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
3
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
7
>>>
```

Whenever we want to add randomness to our games, we can use the `randint()` function. And we use randomness in most games. (Think of how many board games use dice.)

You can also try out different ranges of numbers by changing the arguments. Type

random.randint(1, 4) to only get integers between 1 and 4 (including both 1 and 4). Or try random.randint(1000, 2000) to get integers between 1000 and 2000.

```
>>> random.randint(1, 4)
3
>>> random.randint(1, 4)
4
>>> random.randint(1, 4)
1
>>> random.randint(1, 4)
3
>>> random.randint(1000, 2000)
1294
>>> random.randint(1000, 2000)
1585
>>> random.randint(1000, 2000)
1000
>>> random.randint(1000, 2000)
1971
>>>
```

Be sure to type random.randint(1, 20) and not randint(1, 20), otherwise the computer will not know to look inside the random module for the randint() function. Then it will show you an error like below:

```
>>> randint(1, 20)

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    randint(1, 20)
NameError: name 'randint' is not defined
>>>
```

Remember, your program needs to run import random before it can call the random.randint() function. This is why import statements usually go at the beginning of the program.

Arguments

The integers between the parentheses in the random.randint(1, 20) function call are called **arguments**. Some functions require that you pass them values when you call them. Look at these function calls:

```
raw_input()
random.randint(1, 20)
```

The raw_input() function has no arguments. The randint() function has two arguments. When we have more than one argument, we separate them by putting commas in between the

arguments. Programmers say that the arguments are **delimited** (that is, separated) by commas. This is how the computer knows when one value ends and another begins.

If you pass too many or too few arguments in a function call, Python will display an error message. In the picture below, we first called `randint()` with only one argument (too few), and then we called `randint()` with three arguments (too many).

```
>>> random.randint(1)

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    random.randint(1)
TypeError: randint() takes exactly 3 arguments (2 given)
>>> random.randint(1, 2, 3)

Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    random.randint(1, 2, 3)
TypeError: randint() takes exactly 3 arguments (4 given)
>>> 
```

Code Explanation continued...

```
10. print 'Well, ' + myName + ', I am thinking of a number
    between 1 and 20.'
```

This `print` statement welcomes the player by name, and tells them that the computer is thinking of a random number. Remember how I said the `print` statement only takes one string? It does. Look at the line carefully. The plus signs concatenate the three strings to evaluate down to one string, and that is the one string for the `print` statement. It might look like the commas are separating the strings, but if you look closely you see that the commas are *inside* of the quotes and part of the strings themselves.

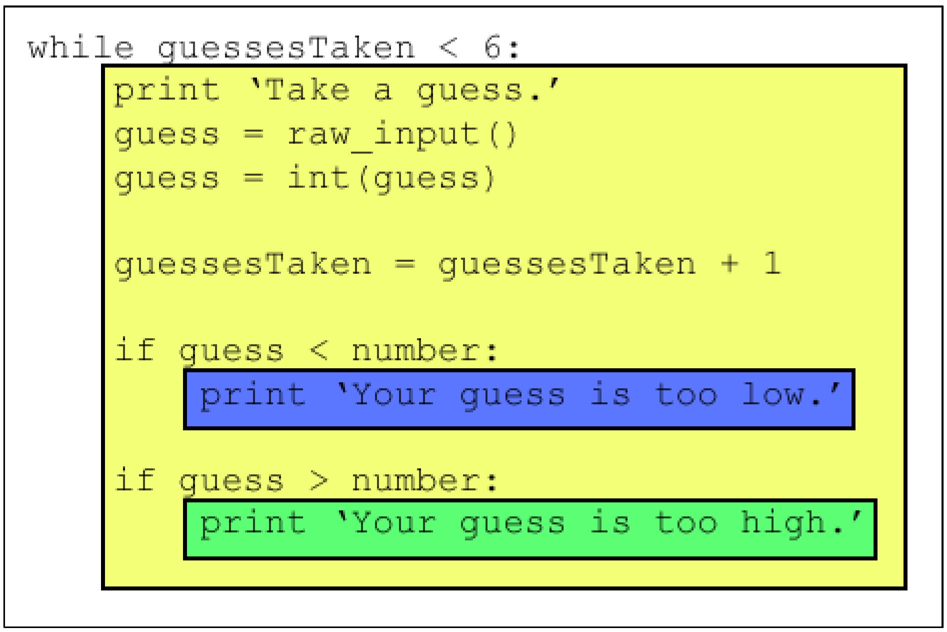
```
12. while guessesTaken < 6:
```

This is a **while statement**. Like `import`, it has a special keyword to Python. The `while` statement is made up of the `while` keyword, followed by an expression, followed by a colon (the : sign). The next line after the `while` statement is the beginning of a while-block. The while-block is made up of the lines of code that have at least 4 or more spaces in front of it (which are lines 13 through 26). The

expression next to the while keyword is also called a **condition**. Before we understand what is going on with this code, let's learn about blocks and conditions.

Blocks

A **block** is made up of several lines of code grouped together. You can tell when the block begins and ends by looking at the line's **indentation** (that is, the number of spaces in front of the line). The block starts when the indentation of a line of code is *more than the previous line*. The block ends when the indentation *returns to what it was before the block started*. It is easier to see with a picture. This picture has each block highlighted with a different color:



The lines of code inside the yellow box are all in the same block. Because this block follows the `while` statement, we call it a while-block. Blocks can contain other blocks. Notice that the yellow block contains the blue and green blocks. The blue and green blocks are still blocks, even though they only have one line of code and are inside another block. The Python interpreter knows when a block is finished because a line of code will have the same indentation before the block started.

It is important to get the indentation correct. Usually the indentation of a block is four spaces. The indentation of a block *inside another block* is eight spaces. And the indentation of a block inside a block inside a block is twelve spaces. Notice that when we type code into IDLE, each letter is the same width. You can look at how many letters are on the line above or below to see how many spaces you have put in.

The indentation doesn't have to be four spaces more than the last indentation, but that is the convention (that is, the usual way of doing things) in the Python language.

Here is a picture of that same code, except now we have red boxes for each space to make it easier to count the spaces. The yellow block includes all the lines with at least four spaces in front. The blue block is the first line with eight spaces of indentation. The green block is the second line with eight

spaces of indentation. Because there is a line with smaller indentation after the blue block, we know that the blue block has ended. This is why the blue and green blocks are separate blocks.

```
while guessesTaken < 6:
    print ‘Take a guess.’
    guess = raw_input()
    guess = int(guess)

    guessesTaken = guessesTaken + 1

    if guess < number:
        print ‘Your guess is too low.’

    if guess > number:
        print ‘Your guess is too high.’
```

We call the block after the `while` keyword a loop block because when the program reaches the bottom of the block, it will loop back to the top. Then it rechecks if the condition is still true. If it is, our program enters the loop block again. If the condition is false, then our program jumps down to the line after the loop block. The loop block is also called a while-block, because it starts with the while keyword. You can learn what it means for a condition to be true or false in the next section.

Conditions (a special kind of expression) and Booleans (a new data type)

Remember we were talking about the line of code with the `while` statement:

```
12. while guessesTaken < 6:
```

I called the expression that came after the `while` keyword the condition. How do we know it is an expression? Because it contains two values (the value in the variable `guessesTaken`, and the integer value `6`) connected by an operator (the `<` sign, which is called the "less than" sign). This is a new type of operator called a **comparison operator**. Expressions with comparison operators won't evaluate to an integer or a string, but a new data type called a **boolean**.

What's a boolean? Well, for the integer data type, there are many different integer values we can have:

```
4
99
0
1236892892
```

And for the string data type, there are also many different string values we can have:

```
'Hello world!'
'My name is Albert.'
'fhsu$$iwehiu^4tihggs@is34
'42'
```

But for the boolean data type, there are two and only two values:

```
True
False
```

When you type one of these values into your program, remember that they are case-sensitive. You must type `True` or `False`, not `true` or `TRUE` or `fAlSe`. Boolean values are not string values, so you do not put a ' quote character around them.

A **condition** is an expression that uses comparison operators (such as the `<` "less than" sign). Conditions will always evaluate to a boolean value. This is like how expressions with math operators (like + or - or *) will evaluate to integers.

Let's look at the condition in our code: `guessesTaken < 6`

What this translates to is "is the value stored in `guessesTaken` less than the value `6`?" If it is, then the condition evaluates to `True`. If it does not, then the condition evaluates to `False`. Remember in line 4, we stored the value `0` in `guessesTaken`. So this condition is asking, "is the value 0 less than the value 6". We know that this is true, so the condition evaluates to the boolean value of `True`.

Let's go back to the interactive shell for a bit. Type in the following conditions (which are also expressions):

```
0 < 6
6 < 0
50<10
10 <        11
10 < 10
```

```
>>> 0 < 6
True
>>> 6 < 0
False
>>> 50<10
False
>>> 10 <          11
True
>>> 10 < 10
False
>>>
```

The condition `0 < 6` returns the boolean value `True` because the number 0 is less than the number 6. But because 6 is not less than 0, the condition `6 < 0` evaluates to `False`. 50 is not less than 10, so `50<10` is `False`. 10 is less than 11, so `10 <          11` is `True`.

But what about `10 < 10`? Why does it evaluate to `False`? Because the number 10 is not smaller than the number 10. They are exactly the same size. If Alice was the same height as Bob, if would be false to say that Alice was shorter than Bob. Likewise, `10 < 10` evaluates to `False`.

There are some other comparison operators besides `<`. Here they are:

Operator Sign	Operator Name
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`==`	Equal to
`!=`	Not equal to

Let's try typing some conditions into the shell to see how these operators work:

```
0 > 6
6 > 0
10 > 10
10 == 10
10 == 11
11 == 10
10 != 10
10 != 11
'Hello' == 'Hello'
'Hello' == 'Good bye'
'Hello' == 'HELLO'
'Good bye' != 'Hello'
```

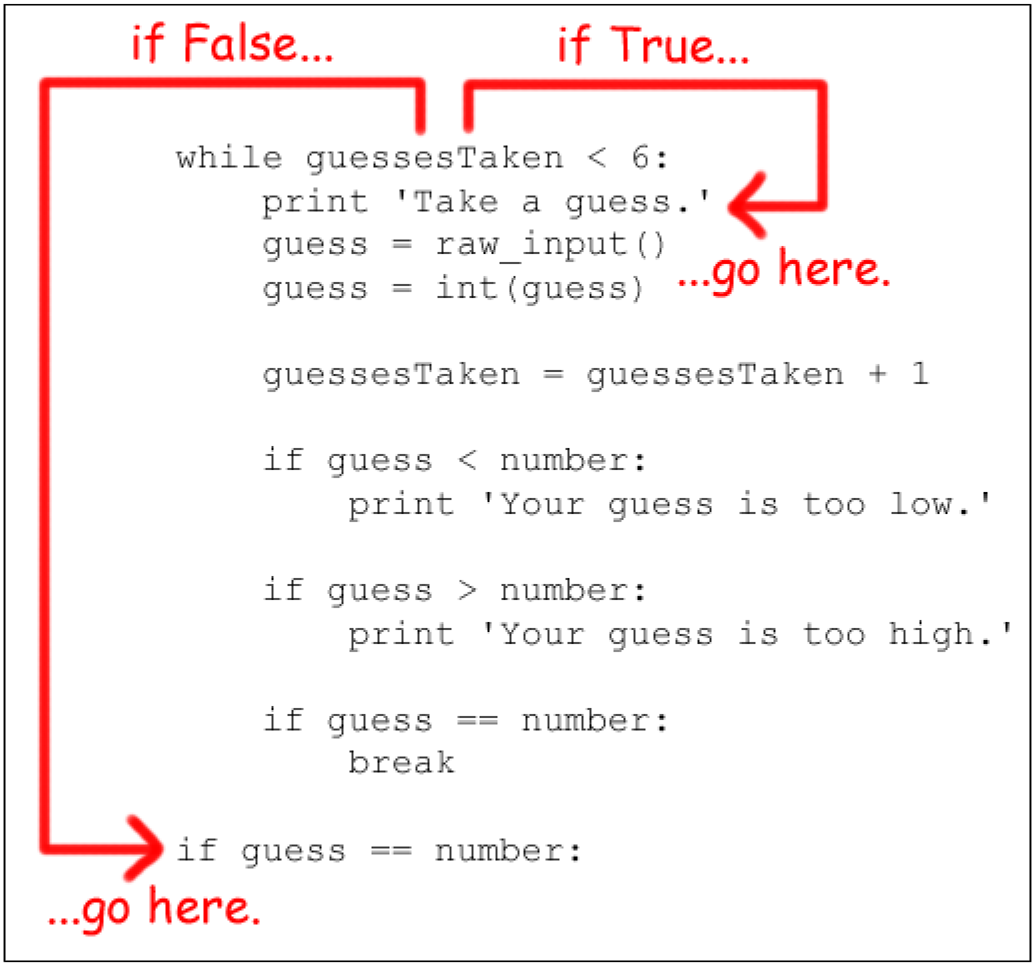
```
>>> 0 > 6
False
>>> 6 > 0
True
>>> 10 > 10
False
>>> 10 == 10
True
>>> 10 == 11
False
>>> 11 == 10
False
>>> 10 != 10
False
>>> 10 != 11
True
>>> 'Hello' == 'Hello'
True
>>> 'Hello' == 'Good bye'
False
>>> 'Hello' == 'HELLO'
False
>>> 'Good bye' != 'Hello'
True
>>>
```

Notice that there is a difference between the assignment operator (the `=` sign) and the "equal to" comparison operator (the `==` sign). The `=` sign is used to assign a value to a variable, and the `==` sign is used in expressions to see if two values are the same or not. It's easy to accidentally use one when you meant to use the other, so be careful of what you type in.

Now that we have covered what conditions, comparison operators, and booleans are, let's talk about what `while` statements do.

```
12. while guessesTaken < 6:
```

The `while` statement marks the beginning of a **loop**. Sometimes in our programs, we want the program to do something over and over again. When the execution reaches a `while` statement, it evaluates the condition next to the `while` keyword. If the condition evaluates to `True`, the execution moves inside the while-block. (In our program, the while-block begins on line 13.) If the condition evaluates to `False`, the execution moves past the while-block. (In our program, the first line after the while-block is line 28.)



Let's say the condition evaluates to `True` (which it does the first time, because the value of `guessesTaken` is `0`.) Execution will enter the while-block at line 13 and keep going down. After the reaches the end of the while-block, instead of going down to the next line, it jumps back up to the `while` statement's line (line 12). It then re-evaluates the condition, and if it is `True` then we enter the while-block again.

This is how the loop works. As long as the condition is `True`, we will keep executing the code inside the while-block over and over again until we reach the end of the while-block and the condition is `False`. So until `guessesTaken` is equal to or greater than `6`, we will keep looping. Think of the `while` statement as saying, "while this condition is true, keep looping through the code in this block".

Code Explanation continued...

```
13.     print 'Take a guess.' # There are four spaces in
   front of print.
14.     guess = raw_input()
```

The program will now ask us for a guess. We type in what we guess the number is, and then this is stored in a variable named `guess`.

```
15.     guess = int(guess)
```

Here we call a new function called `int()`. The `int()` function takes one argument. The `raw_input()` function returned a string of text that player typed. But in our program, we will want an integer, not a string. Remember that Python considers the string '5' and the integer 5 to be different values. So the `int()` function will take the value we give it and return the integer form of it.

Let's play around with the `int()` function in the interactive shell. Try typing the following:

```
int('42')
int(42)
int('hello')
int('forty-two')
int('    42    ')
2 + int('2')
```

```
>>> int('42')
42
>>> int(42)
42
>>> int('hello')

Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    int('hello')
ValueError: invalid literal for int() with base 10: 'hello'
>>> int('forty-two')

Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    int('forty-two')
ValueError: invalid literal for int() with base 10: 'forty-two'
>>> int('    42    ')
42
>>> 2 + int('2')
4
>>> 
```

We can see that the `int('42')` call will return the integer value 42. The `int(42)` will also do this

(though it is kind of pointless to convert an integer to an integer). However, even though you can pass a string to the `int()` function, you cannot just pass any string. Passing `'hello'` to `int()` (like we do in the `int('hello')` call) will result in an error. The string we pass to `int()` must be made up of numbers.

And the integer we pass to `int()` must be in numbers, it cannot be written out. This is why `int('forty-two')` also fails and produces an error. The `int()` function is slightly forgiving, because if our string has spaces on either side, it will still run without error. (This is why the `int('    42    ')` call works.)

The `2 + int('2')` line shows an expression that adds an integer 2 to the return value of `int('2')` (which evaluates to 2 as well). The expression evaluates to 2 + 2, which then evaluates to 4. So even though we cannot add an integer and a string (`2 + '2'` would show us an error), we can add an integer to a string that has been converted to an integer.

The `guess` variable originally held the string of what the player typed. We will overwrite the string value stored in guess with the integer value returned by the `int` function.

In our Guess the Number game, if the player typed in something that was not a number, then the function call `int` would result in an error and the program would crash. In the other games in this book, we will add some more code to check for error conditions like this and give the player another chance to enter a correct response.

```
    guessesTaken = guessesTaken + 1
```

Now that the player has taken a guess, we want to increase the number of guesses that we remember the player taking. The first time we enter the loop block, `guessesTaken` has the value of 0. Python will take this value and add 1 to it. 0 + 1 is 1. Then Python will store the new value of 1 to `guessesTaken`. After this line, the value of `guessesTaken` will be 1 more than it was previously.

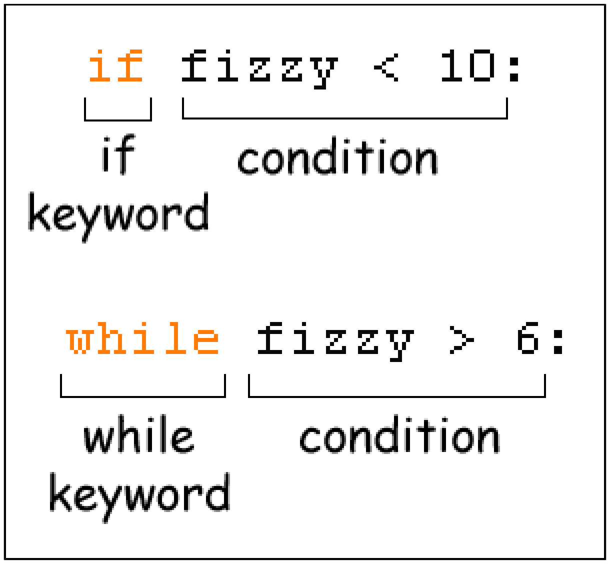
It is easy to think of this line as meaning, "the `guessesTaken` variable should be one more than what it already is". When we add one to a value, programmers say they are **incrementing** the value (because it is increasing by one). When we subtract one from a value, programmers say they are **decrementing** the value (because it is decreasing by one).

`if` Statements

```
    if guess < number:
        print 'Your guess is too low.' # There are
```

```
    eight spaces in front of print.
```

This is called an `if` statement. It has a new keyword, `if`. Next to the `if` keyword is the condition. The block that follows the `if` keyword is called an if-block. The `if` statement is very similar to the `while` statement. They both have a keyword, followed by a condition, and then a block of code.



The `if` statement works almost the same way as a `while` statement, too. If the condition is `True`, then execution enters the if-block. If the condition is `False`, then the execution skips past the if-block. Unlike the while-block, execution does not jump back to the `if` statement at the end of the if-block. It just continues on down to the next line.

This `print` statement is the only line inside the if-block. If the integer the player typed is less than the random integer the computer thought up, then we will display to the player `"Your guess is too low."` If the integer the player entered is equal to or larger than the random integer (in which case, the condition next to the `if` keyword would have been `False`), then this block would have been skipped over.

Code Explanation continued...

```
    if guess > number:
        print 'Your guess is too high.'
```

Here is another `if` statement. This time, we check if the player's guess is larger than the random integer. If so, we will enter the if-block that follows it. The `print` line tells the player that their guess is

too big.

In case you haven't thought of it, these two conditions cannot both be `True`. The player's guess (which is stored in the `guess` variable) can either be higher OR lower than the computer's guess, but it can not be higher AND lower. This means we will never see both messages at the same time. There is one more case to consider, and that is if the guess is equal to the random integer. We will cover this in the next line.

```
25.     if guess == number:
26.         break
```

This `if` statement's condition checks to see if the guess is equal to the random integer. If it is, we will enter the if-block that follows it.

The line inside the if-block is just a **break** statement. The `break` statement tells the program to immediately jump to the out of the while-block that it is inside of, and to the first line after the end of the while-block. The `while` statement's condition is not rechecked.

The `break` statement is just the `break` keyword by itself, with no condition or colon (the : sign).

If the player's guess is not equal to the random integer, we do not break out of the while-block, we will reach the bottom of the while-block anyway. Once we reach the bottom of the while-block, the program will loop back to the top and recheck the condition (`guessesTaken < 6`). Remember after the `guessesTaken = guessesTaken + 1` line of code executed, the new value of `guessesTaken` is `1`. Because `1` is less than `6`, we enter the loop again.

If the player keeps guessing too low or too high, the value of `guessesTaken` will change to `2`, then `3`, then `4`, then `5`, then `6`. If the player guessed the number correctly, the condition in the `if guess == number` statement would be `True`, and we would have executed the `break` statement. Otherwise, we keep looping. But when guessesTaken has the number `6` stored, the `while` statement's condition is `False`. (`6` is not less than `6`, rather `6` is equal to `6`) Because the `while` statement's condition is `False`, we will not enter the loop and instead jump to the end of the while-block.

```
28. if guess == number:
```

This line of code isn't the same line in line 25. This line has no indentation, and is outside the while-block. When we got out of the `while` block, it was either because the `while` statement's condition was

False (which happens if the player ran out of guesses) or if we executed the break statement (which happens if the player guessed the number right). On this line we recheck if the player guessed correctly, and if so, we enter the if-block that follows.

```
29.     guessesTaken = str(guessesTaken)
```

This line is inside the if-block, and only executes if the condition was True.

This line is like the guess = int(guess) line of code. Here we call the new function str(), which returns the string form of the argument we give it. We want to change the value in guessesTaken (which is an integer) into the string version of itself.

The str() and int() functions are very important, because it is important to know that integers and strings are different data types with different values. The integer 42 and the string '42' are entirely different. But if we ever need to get the value of one data type as a value of another data type, str() and int() can be very handy.

```
30.     print 'Good job, ' + name + '! You guessed my
   number in ' + guessesTaken + ' guesses!'
```

This line is also inside the if-block, and only executes if the condition was True.

This line will tell the player that they have won, and how many guesses it took them. The reason why we had to change the guessesTaken value into a string is because we can only add strings to other strings. If we tried to add a string to an integer, the Python interpreter would get confused and display an error.

```
32. if guess != number:
```

This if statement's condition has a new sign. Just like the == sign means "is equal to", the != sign means "is not equal to". If the value of guess is lower than or higher than (and therefore, not equal to) the random number, then this condition evaluates to True, and we would then enter the block that

follows this `if` statement.

```
    number = str(number)
```

This line is inside the if-block, and only executes if the condition was `True`.

In this block, because the player did not guess the random number we will tell them what it is. But first we will have to store the string version of `number` as the new value of `number`.

```
    print 'Nope. The number I was thinking of was ' +
number
```

This line is also inside the if-block, and only executes if the condition was `True`. This line tells the player what the random number was. At this point, we have reached the end of the source code, so the program terminates.

We've just programmed our first real game! In the last chapter we learned about values and expressions and variables. In this chapter we learned how we can use those along with `if`, `while`, and `break` statements to make the program do different things based on the value of variables or expressions.

Step by Step, One More Time

Let's go over the code one more time. To help you understand everything, I will briefly go through the program just like the computer would, starting from the top. We will remember what the values of variables are ourselves (you can write them down on a piece of paper as we go). This is called **tracing** through the program. It's what programmers do to figure out exactly how the program will behave. Some lines of code are executed more than once (because they are inside loops), so we will go over those lines of code more than once.

```
# This is a guess the number game.
```

This line is a comment. The computer will ignore this line, and move down to line 2.

```
2. import random
```

This line will `import` the `random` module so that we can use the `randint()` function in our program. Line 3 is blank, so the computer will skip ahead to line 4.

```
4. guessesTaken = 0
```

The computer will create a new variable called `guessesTaken`, and the integer `0` will be stored inside this variable.

```
6. print 'Hello! What is your name?'
```

A greeting is displayed to the player.

```
7. myName = raw_input()
```

The `raw_input()` function is called, and will let the user type in a string. This string is then stored in a variable called `myName`. Let's pretend that when the program runs, the player types in Bob. The value of the `myName` variable is the string, `'Bob'`.

```
9. number = random.randint(1, 20)
```

On line 9 we call the `randint()` function, which is inside the `random` module. Because this function is inside a module we imported, we have to put the module name and a period in front of the function name. The two arguments we pass are the integers `1` and `20`. This tells the `randint()` function to return a random integer between 1 and 20 (including 1 and 20). Let's pretend that it returns the integer `8`. The value of `number` will be `8`.

```
10. print 'Well, ' + myName + ', I am thinking of a number
    between 1 and 20.'
```

Because the value inside `myName` is the string `'Bob'`, this will print out `Well, Bob, I am thinking of a number between 1 and 20.`

```
12. while guessesTaken < 6:
```

This is the start of a while-block. If the condition is `True`, then the program execution will enter the while-block. If the condition is `False`, we will skip past the while-block to line 28. The variable `guessesTaken` has `0` stored inside of it, and `0` is less than `6`, which makes the condition `True`. So the next line to run is line 13.

```
13.     print 'Take a guess.' # There are four spaces in
    front of print.
```

We print a message that asks the player to type in a value. There is a comment on this line that the computer ignores. The comment reminds the programmer that we should put four spaces at the beginning of the line because we are now inside a block.

```
14.     guess = raw_input()
```

The player now types in a string, and this string will be stored in the `guess` variable. Let's pretend that the player typed in the string `'12'`.

```
15.     guess = int(guess)
```

We want to store the integer value of what the player typed in, not the string value. `int()` function will return the integer value of the argument we give it. (The argument is the value in between the parentheses next to the function name "int".) The guess variable holds the string `'12'`, so `'12'` is the argument we pass to the `int()` function, and the integer value `12` is what the `int()` function returns. This value is then stored as the new value in the `guess` variable. After this line runs, guess stores the integer `12` instead of the string `'12'`.

```
17.     guessesTaken = guessesTaken + 1
```

The value stored in `guessesTaken` is `0` (this was set on line 4). We want to keep track of how many guesses the player has taken, so we make the new value of `guessesTaken` to be the current value of `guessesTaken` plus one. After this line executes, `guessesTaken` will now hold the integer `1`.

```
19.     if guess < number:
```

Now we check if the if-statement's condition is `True`. The value of `guess` is the integer `12` (set on line 15), and the value of `number` is `8` (set on line 9). `12` is not less than `8`, so this condition is `False`. That means we will skip the if-block that follows and go directly to line 22.

```
22.     if guess > number:
```

This if-statement's condition is `True`, because `12` is larger than `8`, so the program execution enters

the if-block at line 23.

```
23.         print 'Your guess is too high.'
```

We display a message that tells the player their guess was too high.

```
25.     if guess == number:
```

The condition in this if-statement is `False`, because `12` is not equal to `8`. We skip the if-block that follows. But line 28 has fewer spaces than the four spaces we have been indenting our code inside the while-block. That means we have reached the end of the while-block too, and execution will loop back to the while-statement on line 12.

```
12. while guessesTaken < 6:
```

The condition for the while-statement is `True`, because `guessesTaken` is `1`, but `1` is still less than `6`. So the program execution enters the while-block at line 13.

```
13.     print 'Take a guess.' # There are four spaces in
    front of print.
```

We display this message to the player again.

```
14.     guess = raw_input()
```

We get a string typed by the player, and store it in the variable `guess`. Let's pretend that the user typed in the string `'6'`. The string `'6'` is stored in the variable guess, and the old value of `12` is forgotten.

```
15.     guess = int(guess)
```

We want to get the integer value of the string inside `guess`. We pass the `int()` function an argument of `'6'`, and it will return `6`. The new value of `guess` is the integer `6`.

```
17.     guessesTaken = guessesTaken + 1
```

We want to increase the number of guesses taken by one, so the new value of `guessesTaken` is the current value (the integer `1`) plus one. The new value of `guessesTaken` is `2`.

```
19.     if guess < number:
```

We check to see if this if-statement's condition is `True`. It is, because `6` is less than `8`. That means our program's execution will enter the if-block at line 20.

```
20.         print 'Your guess is too low.' # There are
   eight spaces in front of print.
```

We display a message to the player tell them that their guess was too low. The text after the # pound sign is a comment and is ignored.

```
22.     if guess > number:
```

We check if guess (the integer 6) is greater than number (the integer 8). It is not, so this condition is False and we skip the if-block.

```
25.     if guess == number:
```

We check if guess (the integer 6) is equal to number (the integer 8). It is not, so this condition is False and we skip the if-block. We have reached the end of the while-block, so we jump back to line 12.

```
12. while guessesTaken < 6:
```

This time when we check the condition, guessesTaken has the value 3. But 3 is still less than 6, so the condition is True and we enter the while-block again.

```
13.     print 'Take a guess.' # There are four spaces in
    front of print.
```

We ask the player to type in a number again.

```
14.     guess = raw_input()
```

The function call to the raw_input() function lets the player type in a string. Let's pretend that the player types in the string '8'. Then the new value of guess is '8'.

```
15.     guess = int(guess)
```

We want to get the integer value of the string inside `guess`. We pass the `int()` function an argument of `'8'`, and it will return `8`. The new value of `guess` is the integer `8`.

```
17.     guessesTaken = guessesTaken + 1
```

We want to increase the number of guesses taken by one, so the new value of `guessesTaken` is the current value (the integer `2`) plus one. The new value of `guessesTaken` is `3`.

```
19.     if guess < number:
```

We check if `guess` (the integer `8`) is less than `number` (the integer `8`). It is not. (If I had 8 apples and you had 8 apples, you would not say I had less apples than you because we have an equal number of apples.) This condition is `False` and we skip the if-block. Next we execute line 22.

```
22.     if guess > number:
```

We check if `guess` (the integer `8`) is greater than `number` (the integer `8`). It is not, so this condition is `False` and we skip the if-block. Next we execute line 25.

```
25.     if guess == number:
```

We check if `guess` (the integer `8`) is equal than `number` (the integer `8`). It is, so we enter the if-block at line 26.

```
26.         break
```

The `break` statement tells us to break out of the while-block that we are inside, and go to the first line after the while-block. This will be line 28.

```
28. if guess == number:
```

We check if `guess` (the integer `8`) is equal than `number` (the integer `8`). It is, so we enter the if-block at line 29.

```
29.     guessesTaken = str(guessesTaken)
```

On this line we convert `guessesTaken` to the string `'3'`.

```
30.     print 'Good job, ' + myName + '! You guessed my
    number in ' + guessesTaken + ' guesses!'
```

Now we display the winning message to the player. The variable `myName` holds the string value `'Bob'` and `guesses` holds the string value `'3'`, so the final string printed is `'Good job, Bob! You guessed my number in 3 guesses!'`

```
32. if guess != number:
```

This condition will evaluate to `False`, so we skip past the if-block that follows it. But there is no more code after it, so the program terminates.

Some Changes We Could Make

This has been our first game! It was kind of long to go through everything and a lot to learn, but now you are a real game programmer. Just for fun, try changing this program to change the way the game behaves.

For example, you can change these lines:

```
number = random.randint(1, 20)
print 'Well, ' + name + ', I am thinking of a number between 1 and 20.'
```

into these lines:

```
number = random.randint(1, 100)
print 'Well, ' + name + ', I am thinking of a number between 1 and 100.'
```

and now the computer will think of an integer between `1` and `100`.

Or you can change this line:

```
while guessesTaken < 6:
```

into this line:

```
while guessesTaken < 4:
```

and now the player only gets four guesses instead of six guesses.

What Exactly is Programming?

If someone asked you, "What exactly is programming anyway?" what could you say back to them? Programming is just the action of writing the code for programs, that is, creating programs that can be executed by a computer.

"But what exactly is a program?" When you see someone using a computer program (for example, playing our Guess The Number game), all you see is some text appearing on the screen. The program decides what exact text to show on the screen (which is called the **output**), based on its instructions and on the text that the player typed on the keyboard (which is called the **input**). The program has very specific instructions on what text to show the user. A program is a collection of instructions.

"What kind of instructions?" There are only a few different kinds of instructions, really.

- Expressions, which are made up of values connected by operators. Expressions are all evaluated down to a single value, like `2 + 2` evaluates to `4` or `'Hello' + ' ' + 'World'` evaluates to `'Hello World'`. Function calls are also part of expressions because they evaluate to a single value themselves, and this value can be connected by operators to other values. When expressions are next to the `if` and `while` keywords, we also call them conditions.
- Assignment statements, which simply store values in variables so we can remember the values later in our program.
- `if`, `while` and `break` are **flow control statements** because they decide which instructions are executed. The normal flow of execution for a program is to start at the top and execute each instruction going down one by one. But these flow control statements can cause the flow to skip instructions, loop over instructions, or break out of loops. Function calls also change the flow of execution by jumping to the start of a function.
- The `print` statement, which displays text on the screen. Also, the `raw_input()` function can get text from the user through the keyboard. This is called **I/O** (pronounced like the letters, "eye-oh"), because it deals with the input and output of the program.

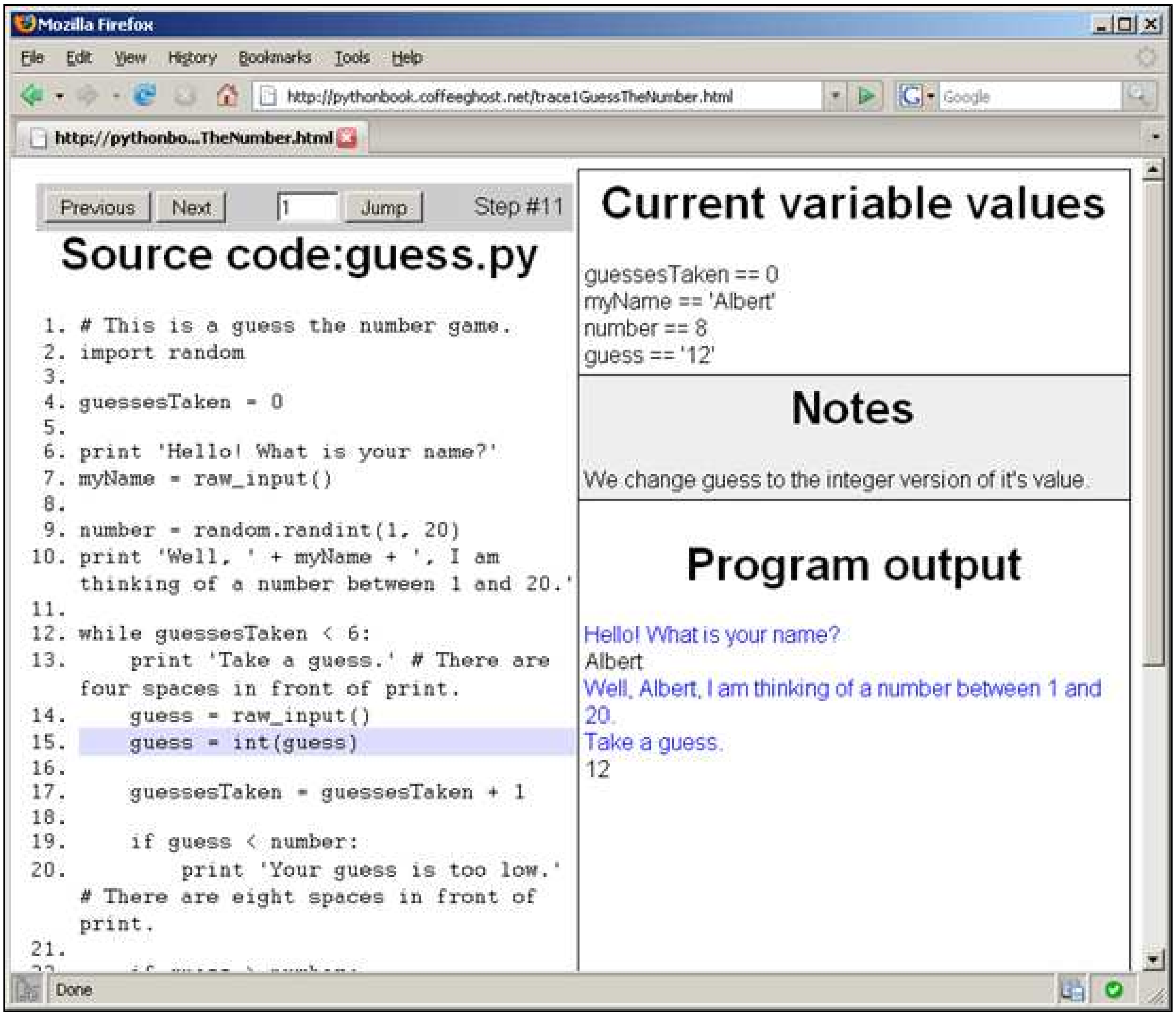
And that's it, just those four things. Of course, there are many details about those four types of instructions. In this book you will learn about new data types and operators, new flow control statements besides `if`, `while` and `break`, and several new functions. There are also different types of I/O (input from the mouse or files on the hard drive, and outputting sound and graphics and pictures instead of just text.)

For the person using your programs, they really only care about that last type, I/O. The user types on the keyboard and then sees things on the screen or hears things from the speakers. But for the computer to figure out what sights to show and what sounds to play, it needs a program, and programs are just a bunch of instructions that you, the programmer, have written.

A Web Page for Program Tracing

If you have access to the Internet and a web browser, you can go to these web sites and see a "visual

tracing" web page that will show each step of the program. This might make it more clear what the Guess the Number program does.



The left side of the web page shows the source code, and the highlighted line is the line of code that is about to be executed. You execute this line and move to the next line by clicking the "Next" button. You can also go back a step by clicking the "Previous" button, or jump directly to a step by typing it in the white box and clicking the "Jump" button.

On the right side of the web page, there are three sections. The "Current variable values" section show you each variable that has been assigned a value, along with the value itself. The "Notes" section will give you a hint about what is happening on the highlighted line. The "Program output" section shows the output from the program, and the input that is sent to the program. (This web page automatically enters text to the program when the program asks.)

So go to each of these web pages and click the "Next" and "Previous" buttons to trace through the program like we did above.

- Guess the Number, trace 1 - http://pythonbook.coffeeghost.net/book1/traces/trace1GuessTheNumber.html
- Guess the Number, trace 2 - http://pythonbook.coffeeghost.net/book1/traces/trace2GuessTheNumber.html

Things Covered In This Chapter:

- `import` statements
- Modules
- Arguments
- `while` statements
- Conditions
- Blocks
- Comparison operators
- The difference between `=` and `==`.
- `if` statements
- The `break` keyword.
- The `str()` function.
- The `random.randint()` function.

Chapter 3 - Jokes

How Programs Run on Computers

Now we will write a program to tell jokes to the user. Before we go into the code, you should know how your programs run on the computer.

The computer you use runs a very large program, called an **operating system**. Your operating system (called an **OS** (pronounced like the letters, "oh-ess") for short) may be Windows, MacOS, Linux or another one. The OS is a program that runs other programs called **applications** like a web browser, word processor, email client, or computer games. The OS makes it easy for programmers to write applications and games that can run on computers made up of different hardware.

Hardware includes the parts of the computer that you can touch (the monitor, or the keyboard and mouse, or a printer). **Software** is another name for programs like the OS or applications or games that run on the computer. Think of the computer as if it were a book. The book's hardware would be the cover and paper pages and even the ink on the page (the things you can touch.) The book's software would be the story and characters that the book describes. Using software or playing games that were made by someone else is like reading a book of stories that was written by another person. But writing software (such as your own games) is like writing your own stories.

It would be very difficult for programmers to make their programs run on several different pieces of hardware. For example, when you write your games, you don't need to know how to make text appear on all the different monitors made by all the different companies in the world. Your program just has a `print` statement, which tells the OS to figure out how to make it appear on the monitor no matter what brand or type of monitor the user has.

The OS makes running programs easy on us, but it still only knows a language called **machine code**. Machine code has some very, very, very basic instructions that are simple enough for computer's main microchip (called the **CPU** ("see-pee-you"), or **Central Processing Unit**) to understand. Writing programs in the machine code language is very long and boring.

Machine code is written in ones and zeros and look like pages and pages of this: 10101101 00110000 11000000. These instructions aren't very easy for humans to work with. **Assembly language** gives instructions names like MOV, JMP, PUSH, or XOR. This makes reading and writing the instructions easier but putting them together in a program is still long and complicated.

This is where **higher-level programming languages** come in. High-level languages include Python, Java, C++, Pascal, Perl, Basic, and many others. These languages take care of many of the details of machine code. A programmer writes her program in a higher-level language like Python, and then a program called the **interpreter** translates this language into machine code that the computer executes. Even though our "Hello world!" program was just one line long when written in Python, in machine code it would be several hundred or a few thousand lines.

The interpreter is the program you downloaded from http://www.python.org and installed in chapter one. That download also included a program called IDLE, which is the program we type our code into.

When we run a program in the file editor, or type an instruction into the interactive shell, the IDLE program sends that source code to the Python interpreter for translation. The interpreter translates it into machine code, and then the CPU understands how to run the program.

Whew! That was a lot of information. As computers get faster and faster, they also become more and more complicated. To manage all of this complexity, programmers started to write programs that would help them write new programs! One of these programs is the Python interpreter that you are using. This is kind of like using a stone axe to help build a hammer, and then a hammer to help build a electric drill and other power tools, and then using those power tools to build a large bulldozer.

The reason I am explaining all of this is so that you understand that when you write code in Python, it is being passed to another program called the Python interpreter, which then translates it so that the operating system and computer can run your code.

This next program is simpler compared to the "Guess the Number" game in chapter two. Open a new file editor window by clicking on File, then clicking on New Window and enter this source code:

Sample Run

```
What do you get when you cross a snowman with a vampire?

Frostbite!

What do dentists call an astronaut's cavity?

A black hole!

Knock knock.

Who's there?

Interrupting cow.

Interrupting cow wh MOO!
```

Source Code

jokes.py

```
print 'What do you get when you cross a snowman with a vampire?'
raw_input()
print 'Frostbite!'
print
print 'What do dentists call a astronaut\'s cavity?'
```

```
raw_input()
print 'A black hole!'
print
print 'Knock knock.'
raw_input()
print "Who's there?"
raw_input()
print 'Interrupting cow.'
raw_input()
print 'Interrupting cow wh',
print 'MOO!'
```

Don't worry if you don't understand everything in the program. Just save and run the program.

Code Explanation

Let's look at the code more carefully.

```
print 'What do you get when you cross a snowman with a vampire?'
raw_input()
print 'Frostbite!'
print
```

Here we have three `print` statements. Because we don't want to tell the player what the joke's punch line is, we have a call to the `raw_input()` function after the first `print` statement. The player can read the first line, press Enter, and then read the punch line.

The user can still type in a string and hit Enter, but because we aren't storing this string in any variable, the program will just forget about it and move to the next line of code.

The last call to the `print` statement has no string. This tells the program to just print a blank line. Blank lines can be useful to keep our text from being bunched up together.

```
print 'What do dentists call a astronaut\'s cavity?'
raw_input()
print 'A black hole!'
print
```

In the first `print` statement above, you'll notice that we have a slash right before the single quote (that s, the apostrophe). This backslash (\ is a backslash, / is a forward slash) tells us that the letter right after t is an **escape character**. An escape character helps us print out letters that are hard to enter into the source code. There are several different escape characters, but in our `print` statement the escape character is the single quote.

We have to have the single quote escape character because otherwise the Python interpreter would hink that this quote meant the end of the string. But we want this quote to be a part of the string. When we print this string, the backslash will not show up.

Some Other Escape Characters

What if you really want to display a backslash? This line of code would not work:

```
print 'He flew away in a green\teal helicopter.'
```

That print statement would show up as:

```
He flew away in a green    eal helicopter.
```

This is because the "t" in "teal" was seen as an escape character since it came after a backslash. The escape character t simulates pushing the Tab key on your keyboard. Escape characters are there so that strings can have characters that cannot be typed in.

Instead, try this line:

```
print 'He flew away in a green\\teal helicopter.'
```

Here is a list of escape characters in Python:

Escape Characters	
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\n	Newline
\t	Tab

Quotes and Double Quotes

Strings don't always have to be in between single quotes in Python. You can also put them in between double quotes. These two lines print the same thing:

```
print 'Hello world'
print "Hello world"
```

But you cannot mix quotes. This line will give you an error if you try to use them:

```
print 'Hello world"
```

```
>>> print 'Hello world"
SyntaxError: EOL while scanning single-quoted string
>>> 
```

I like to use single quotes because I don't have to hold down the shift key on the keyboard to type them. It's easier to type, and the computer doesn't care either way.

But remember, just like you have to use the escape character \' to have a single quote in a string surrounded by single quotes, you need the escape character \" to have a double quote in a string surrounded by double quotes. For example, look at these two lines:

```
print 'I asked to borrow Abe\'s car for a week. He said, "Sure."'
print "He said, \"I can't believe you let him borrow your car.\""
```

```
>>> print 'I asked to borrow Abe\'s car for a week. He said, "Sure."'
I asked to borrow Abe's car for a week. He said, "Sure."
>>> print "He said, \"I can't believe you let him borrow your car.\""
He said, "I can't believe you let him borrow your car."
>>> 
```

Did you notice that in the single quote strings you do not need to escape double quotes, and in the double quote strings you do not need to escape single quotes? The Python interpreter is smart enough to know that if a string starts with one type of quote, the other type of quote doesn't mean the string is ending.

Code Explanation continued...

```
print 'Knock knock.'
raw_input()
print "Who's there?"
raw_input()
print 'Interrupting cow.'
raw_input()
print 'Interrupting cow wh',
```

```
print 'MOO!'
```

Did you notice the comma at the end of the second to last string? Normally, `print` adds a newline character to the end of the string it prints. (This is why a blank `print` statement will just print a newline.) This comma means we do not want to `print` a newline at the end. This is why `'MOO!'` appears next to the previous line, instead of on its own line.

Things Covered In This Chapter

- Using `print` with no parameters to display blank lines.
- Escape characters.
- Using single quotes and double quotes for strings.
- Using commas at the end of print statements.

Chapter 4 - Dragon Realm

In this game, the player is in a land full of dragons. The dragons all live in caves with their large piles of collected treasure. Some dragons are friendly, and will share their treasure with you. Other dragons are greedy and hungry, and will eat anyone who enters their cave. The player is in front of two caves, one with a friendly dragon and the other with a hungry dragon. The player is given a choice between the two.

Open a new file editor window by clicking on the File menu, then click on New Window. In the blank window that appears type in the source code and save the source code as dragon.py. Then run the program by pressing F5.

Sample Run

```
You are in a land full of dragons. In front of you,
you see two caves. In one cave, the dragon is friendly
and will share his treasure with you. The other dragon
is greedy and hungry, and will eat you on sight.

Which cave will you go into? (1 or 2)
1
You approach the cave...
It is dark and spooky...
A large dragon jumps out in front of you! He opens his jaws
and...

Gobbles you down in one bite!
Do you want to play again? (yes or no)
no
```

Source Code

dragon.py

```
import random
import time

def displayIntro():
    print 'You are on a planet full of dragons. In front of you,'
    print 'you see two caves. In one cave, the dragon is friendly'
    print 'and will share his treasure with you. The other dragon'
```

```
    print 'is greedy and hungry, and will eat you on sight.'
    print

def chooseCave():
    cave = ''
    while cave != '1' and cave != '2':
        print 'Which cave will you go into? (1 or 2)'
        cave = raw_input()

    return cave

def checkCave(chosenCave):
    print 'You approach the cave...'
    time.sleep(2)
    print 'It is dark and spooky...'
    time.sleep(2)
    print 'A large dragon jumps out in front of you! He opens his jaws and...'
    print
    time.sleep(2)

    friendlyCave = random.randint(1, 2)

    if chosenCave == str(friendlyCave):
        print 'Gives you his treasure!'
    else:
        print 'Gobbles you down in one bite!'

playAgain = 'yes'
while playAgain == 'yes' or playAgain == 'y':

    displayIntro()

    caveNumber = chooseCave()

    checkCave(caveNumber)

    print 'Do you want to play again? (yes or no)'
    playAgain = raw_input()
```

Code Explanation

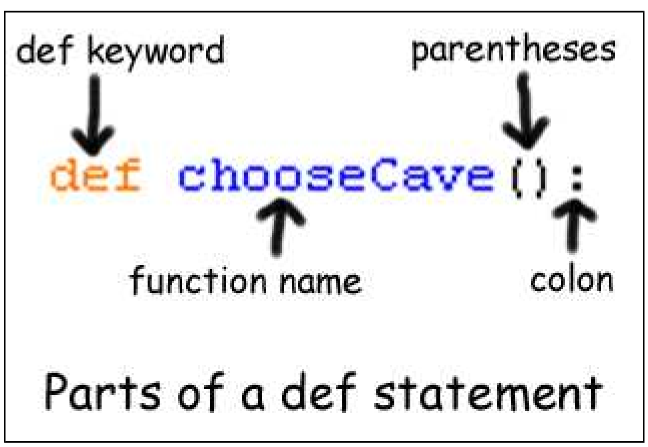
Let's look at the source code in more detail.

```
import random
import time
```

Here we have two `import` statements. We import the `random` module like we did in the Guess the Number game. In Dragon Realm, we will also want some time-related functions that the `time` module includes, so we will import that as well.

```
def displayIntro():
    print 'You are on a planet full of dragons. In front of you,'
    print 'you see two caves. In one cave, the dragon is friendly'
    print 'and will share his treasure with you. The other dragon'
    print 'is greedy and hungry, and will eat you on sight.'
    print
```

Here is a new type of statement, the **def** statement. The `def` statement is made up of the `def` keyword, followed by a function name with parentheses, and then a colon (the : sign). There is a block after the statement called the def-block.



Parts of a def statement

def Statements

The `def` statement isn't a call to a function named `displayIntro()`. Instead, the `def` statement

means we are creating, or **defining**, a new function that we can call later in our program. After we define this function, we can call it the same way we call other functions. When we call this function, the code inside the def-block will be executed.

We also say we define variables when we create them with an assignment statement. The code `spam = 42` defines the variable `spam`.

Remember, the def statement doesn't execute the code right now, it only defines what code is executed when we call the `displayIntro()` function later in the program. When the program's execution reaches a `def` statement, it skips down to the end of the def-block. We will jump back to the top of the def-block when the `displayIntro()` function is called. It will then execute all the print statements inside the def-block. So we call this function when we want to display the "You are on a planet full of dragons..." introduction to the user.

When we call the `displayIntro()` function, the program's execution jumps to the start of the function on line 5. When the function's block ends, the program's execution returns to the line that called the function.

```
11. def chooseCave():
```

Here we are defining another function called `chooseCave`. The code in this function will prompt the user to select which cave they should go into.

```
12.     cave = ''
13.     while cave != '1' and cave != '2':
```

Inside the `chooseCave()` function, we create a new variable called `cave` and store a blank string in it. Then we will start a `while` loop. This while statement's condition contains a new operator we haven't seen before called `and`. Just like the `-` or `*` are mathematical operators, and `==` or `!=` are comparison operators, the `and` operator is a **boolean operator**.

Boolean Operators

Boolean logic deals with things that are either true or false. This is why the boolean data type only has two values, `True` and `False`. Boolean statements are *always* either true or false. If the statement is not true, then it is false. And if the statement is not false, then it is true.

Do you remember how the `*` operator will combine two integer values and produce a new integer value (the product of the two original integers)? And do you also remember how the `+` operator can combine two strings and produce a new string value (the concatenation of the two original strings)? The `and` operator combines two boolean values to produce a new boolean value. Here's how the `and` operator works.

Think of the sentence, "Cats have whiskers and dogs have tails." This sentence is true, because "cats have whiskers" is true and "dogs have tails" is also true.

But the sentence, "Cats have whiskers and dogs have wings." would be false. Even though "cats have whiskers" is true, dogs do not have wings, so "dogs have wings" is false. The entire sentence is only true if both parts are true because the two parts are connected by the word "and." If one or both parts are false, then the entire sentence is false.

The `and` operator in Python works this way too. If the boolean values on both sides of the `and` keyword are `True`, then the expression with the `and` operator evaluates to `True`. If either of the boolean values are `False`, or both of the boolean values are `False`, then the expression evaluates to `False`.

So let's look at line 13 again:

```
13.     while cave != '1' and cave != '2':
```

This condition is made up of two expressions connected by the `and` operator. We first evaluate these expressions to get their boolean values. Then we evaluate the boolean values with the `and` operator.

The string value stored in `cave` when we first execute this while statement is the blank string, `''`. The blank string does not equal the string `'1'`, so the left side evaluates to `True`. The blank string also does not equal the string `'2'`, so the right side evaluates to `True`. So the condition then turns into `True and True`. Because both boolean values are `True`, the condition finally evaluates to `True`. And because the while statement's condition is `True`, the program execution enters the while-block.

This is all done by the Python interpreter, but it is important to understand how the interpreter does this. This picture shows the steps of how the interpreter evaluates the condition (if the value of `cave` is the blank string):

```
while cave != '1' and cave != '2':

while '' != '1' and cave != '2':
```

```
while True and cave != '2':

while True and '' != '2':

while True and True:

while True:
```

Try typing the following into the interactive shell:

```
True and True
True and False
False and True
False and False
```

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
>>>
```

There are two other boolean operators. The next one is the `or` operator. The `or` operator works similar to the `and`, except it will evaluate to `True` if EITHER of the two boolean values are `True`. The only time the `or` operator evaluates to `False` is if both of the boolean values are `False`.

The sentence "Cats have whiskers or dogs have wings." is true. Even though dogs don't have wings, when we say "or" we mean that one of the two parts is true. The sentence "Cats have whiskers or dogs have tails." is also true. (Most of the time when we say this OR that, we mean one thing is true but the other thing is false. In programming, "or" means that either of the things are true, or maybe both of the things are true.)

Try typing the following into the interactive shell:

```
True or True
True or False
False or True
False or False
```

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
>>>
```

The third boolean operator is `not`. The `not` operator is different from every other operator we've seen before, because it only works on one value, not two. There is only value on the right side of the `not` keyword, and none on the left. The `not` operator will evaluate to `True` as `False` and will evaluate `False` as `True`.

Try typing the following into the interactive shell:

```
not True
not False
True not
```

```
>>> not True
False
>>> not False
True
>>> True not
SyntaxError: invalid syntax
>>>
```

Notice that if we put the boolean value on the left side of the `not` operator results in a syntax error.

We can use both the `and` and `not` operators in a single expression. Try typing `True and not False` into the shell:

```
>>> True and not False
True
>>>
```

Normally the expression `True and False` would evaluate to `False`. But the `True and not False` expression evaluates to `True`. This is because `not False` evaluates to `True`, which turns the expression into `True and True`, which evaluates to `True`.

If you ever forget how the boolean operators work, you can look at these charts, which are called **truth tables**:

A	**and**	**B**	**is**	**Entire**

				statement
True	and	True	is	True
True	and	False	is	False
False	and	True	is	False
False	and	False	is	False

A	**or**	**B**	**is**	**Entire statement**
True	or	True	is	True
True	or	False	is	True
False	or	True	is	True
False	or	False	is	False

not A	**is**	**Entire statement**
not True	is	False
not False	is	True

Code Explanation continued...

```
        print 'Which cave will you go into? (1 or 2)'
        cave = raw_input()
```

Here, the player is asked to enter which cave they chose to enter by typing in 1 or 2 and hitting enter. Whatever string the player typed will be stored in `cave`. After this code is executed, we jump back to the top of the `while` statement and recheck the condition. Remember that the line was:

```
    while cave != '1' and cave != '2':
```

If this condition evaluates to `True`, we will enter the while-block again and ask the player for a cave number to enter. But if the player typed in 1 or 2, then the `cave` value will either be `'1'` or `'2'`. This causes the condition to evaluate to `False`, and the program execution will continue on past the while loop.

The reason we have a loop here is because the player may have typed in 3 or 4 or HELLO. Our

program doesn't make sense of this, so if the player did not enter 1 or 2, then the program loops back and asks the player again. In fact, the computer will patiently ask the player for the cave number over and over again until the player types in 1 or 2. When the player does that, the while-block's condition will be `False`, and we will jump down past the while-block and continue with the program.

```
17.     return cave
```

This is the **return** keyword, which only appears inside def-blocks. Remember how the `raw_input()` function returns the string value that the player typed in? Or how the `randint()` function will return a random integer value? Our function will also return a value. It returns the string that is stored in `cave`.

This means that if we had a line of code like `spam = chooseCave()`, the code inside `chooseCave()` would be executed and the function call will evaluate to `chooseCave()`'s return value. The return value will either be the string `'1'` or the string `'2'`. (Our `while` loop guarantees that `chooseCave()` will ONLY return either `'1'` or `'2'`.)

The `return` keyword is only found inside def-blocks. Once the `return` statement is executed, we immediately jump out of the def-block. (This is like how the `break` statement will make us jump out of a while-block.) The program execution moves back to the line that had called the function.

You can also use the `return` keyword by itself just to break out of the function, just like the `break` keyword will break out of a `while` loop.

Variable Scope

You should note that the value stored in the `cave` variable in the `chooseCave()` function is forgotten after the execution leaves the function. Just like the values in our program's variables are forgotten after the program ends, variables inside the function are forgotten after the execution leaves the function. Note only that, but when execution is inside the function, we cannot see the variables outside of the function, or variables inside other functions. We call this the variable's **scope**. The only variables that we can use inside a function are the ones we create inside of the function. That is, the scope of the variable is inside in the function's block. The scope of variables created outside of functions is everywhere in the program outside of def-blocks.

Not only that, but if we have a variable named `spam` created outside of a function, if we create a variable named `spam` inside of the function, the Python interpreter will consider them to be two separate variables. That means we can change the value of `spam` inside the function, and this will not change the `spam` variable that is outside of the function. This is because these variables have different scopes.

We have names for these scopes. The scope outside of all functions is called the **global scope**. The

scope inside of a function is called the **local scope**. The entire program has only one global scope, but each function has a local scope of its own.

Variables defined in the global scope can be used outside and inside functions. Variables defined in a function's local scope can only be used inside that function.

When exactly is a variable defined? A variable is defined the first time we use it in an assignment statement. When the program first executes the line:

```
12.     cave = ''
```

...the variable `cave` is defined.

If we call the `chooseCave()` function twice, the value stored in the variable the first time won't be remember the second time around. This is because when the execution left the `chooseCave()` function (that is, left `chooseCave()`'s scope), the `cave` variable was forgotten and destroyed. But it will be defined again when we call the function a second time.

The important thing to remember is that the value of a variable in the local scope is not remembered in between function calls.

Code Explanation continued...

```
19. def checkCave(chosenCave):
```

Now we are defining yet another function named `checkCave()`. Notice that we put the text `chosenCave` in between the parentheses. This is a type of variable called a **parameter**. For some functions, we would pass an argument, like for the `str()` or `randint()` functions:

```
str(guessesTaken)
random.randint(1, 20)
```

When we call `checkCave()`, we will also pass one value to it as an argument. When execution moves inside the `checkCave()` function, a new variable named `chosenCave` will be assigned this value. This is how we pass variable values to functions since functions cannot read variables outside of the function (that is, outside of the function's scope).

Parameters

For example, here is a short program that demonstrates parameters. Imagine we had a short program that looked like this:

```
def sayHello(name):
    print 'Hello, ' + name

print 'Say hello to Alice.'
fizzy = 'Alice'
sayHello(fizzy)
print 'Do not forget to say hello to Bob.'
sayHello('Bob')
```

If we run this program, it would look like this:

```
>>>
Say hello to Alice.
Hello, Alice
Do not forget to say hello to Bob.
Hello, Bob
>>>
```

This program calls a function we have created, `sayHello()` and first passes the value in the `fizzy` variable as an argument to it. (We stored the string `'Alice'` in `fizzy`.) Later, the program calls the `sayHello()` function again, passing the string `'Bob'` as an argument.

The value in the `fizzy` variable and the string `'Bob'` are arguments. We send values as arguments to a function. The variable `name` is a parameter. Parameters are always local variables and only exist inside the function. That is the difference between arguments and parameters. It might be easier to just remember that the thing in between the parentheses in the `def` statement is an argument, and the thing in between the parentheses in the function call is a parameter.

We could have just used the `fizzy` variable inside the `sayHello()` function instead of using a parameter. (This is because the local scope can still see variables in the global scope.) But then we would have to remember to assign the `fizzy` variable a string each time before we call the `sayHello()` function. Parameters make our programs simpler. Look at this code:

```
def sayHello():
    print 'Hello, ' + fizzy

print 'Say hello to Alice.'
fizzy = 'Alice'
sayHello()
print 'Do not forget to say hello to Bob.'
sayHello()
```

When we run this code, it looks like this:

```
>>>
Say hello to Alice.
Hello, Alice
Do not forget to say hello to Bob.
Hello, Alice
>>>
```

This program's `sayHello()` function does not have a parameter, but uses the global variable `fizzy` directly. Remember that you can read global variables inside of functions, you just can't read local variables outside of the function. But now we have to remember to set the `fizzy` variable before calling `sayHello()`. In this program, we forgot to do so, so the second time we called `sayHello()` the value of `fizzy` was still `'Alice'`. Using parameters makes function calling simpler to do, especially when our programs are very big and have many functions.

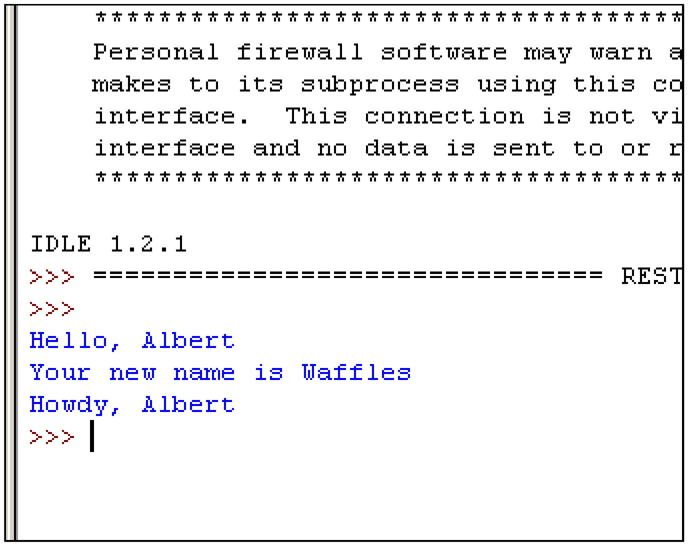
Local Variables and Global Variables with the Same Name

Now look at this program, which is a bit different:

```
def spam(myName):
    print 'Hello, ' + myName
    myName = 'Waffles'
    print 'Your new name is ' + myName

myName = 'Albert'
spam(myName)
print 'Howdy, ' + myName
```

If we run this program, it would look like this:



This program defines a new variable called `myName` and stores the string `'Albert'` in it. Then the program calls the `spam()` function, passing the value in `myName` as an argument. The execution moves to the `spam()` function. The parameter in `spam()` is also named `myName`, and has the argument assigned to it. Remember, the `myName` inside the `spam()` function (the local scope) is considered a different variable than the `myName` variable outside the function (the global scope).

The function then prints `'Hello, Albert'`, and then on the next line changes the value in `myName` to `'Waffles'`. Remember, this only changes the `myName` variable that is inside the function. The `myName` variable that is outside the function still has the value `'Albert'` stored in it.

The function now prints out `'Your new name is Waffles'`, because the `myName` variable in the local scope has changed to `'Waffles'`. The execution has reached the end of the function, so it jumps back down to where the function call was. The local `myName` is destroyed and forgotten. The next line after that is `print 'Howdy, ' + myName`, which will display `Howdy, Albert`.

Remember, the `myName` outside of functions (that is, in the global scope) still has the value `'Albert'`, not `'Waffles'`. This is because the `myName` in the global scope and the `myName` in `spam()`'s local scope are different variables, even though they have the same name.

Where to Put Function Definitions

A function's definition (where we put the `def` statement and the def-block) has to come before you call the function. This is like how you must assign a value to a variable before you can use the variable. If you put the function call before the function definition, you will get an error. Look at this code:

```
sayGoodBye()

def sayGoodBye():
    print 'Good bye!'
```

If you try to run it, Python will give you an error message that looks like this:

```
Traceback (most recent call last):
  File "C:\Python25\test1.py", line 1, in <module>
    sayGoodBye()
NameError: name 'sayGoodBye' is not defined
>>> 
```

To fix this, put the function definition before the function call:

```
def sayGoodBye():
    print 'Good bye!'

sayGoodBye()
```

Code Explanation continued...

```
    print 'You approach the cave...'
    time.sleep(2)
```

We display some text to the player, and then call the `time.sleep()` function. Remember how in our call to `randint()`, the function `randint()` is inside the `random` module? In the Dragon Realm game, we also imported the `time` module. The `time` module has a function called `sleep()` that will pause the program for a few seconds. We pass the integer value `2` as an argument to the `time.sleep()` function to tell it to pause for exactly 2 seconds.

```
    print 'It is dark and spooky...'
    time.sleep(2)
```

Here we print some more text and wait again for another 2 seconds. These short pauses add suspense to the game, instead of displaying all the text all at once. In our jokes program, we called the `raw_input()` function to wait until the player pressed the enter key. Here, the player doesn't have to do anything at all except wait.

```
    print 'A large dragon jumps out in front of you! He
  opens his jaws and...'
    print
    time.sleep(2)
```

We have new action happening in our program. What does the dragon do?

```
    friendlyCave = random.randint(1, 2)
```

Now we are going to have the program randomly chose which cave had the friendly dragon in it. Our call to the `random.randint()` function will return either the integer 1 or the integer 2, and store this value in a variable called `friendlyCave`.

```
    if chosenCave == str(friendlyCave):
        print 'Gives you his treasure!'
```

Here we check if the integer of the cave we chose (`'1'` or `'2'`) is equal to the cave randomly selected to have the friendly dragon. But wait, the value in `chosenCave` was a string (because `raw_input()` returns strings) and the value in `friendlyCave` is an integer (because `random.randint()` returns integers). We can't compare strings and integers with the `==` sign, because they will always be different (`'1'` does not equal `1`).

So we are passing `friendlyCave` to the `str()` function, which returns the string value of `friendlyCave`.

What the condition in this `if` statement is really comparing is the string in `chosenCave` and the string returned by the `str()` function. We could have also had this line instead:

```
if int(chosenCave) == friendlyCave:
```

Then the `if` statement's condition would compare the integer value returned by the `int()` function to the integer value in `friendlyCave`. The return value of the `int()` function is the integer form of the string stored in `chosenCave`.

If the `if` statement's condition evaluates to `True`, we tell the player they have won the treasure.

```
32.     else:
33.         print 'Gobbles you down in one bite!'
```

Line 32 has a is a new keyword. The **else** keyword always comes after the if-block. The else-block that follows the `else` keyword executes if the condition in the `if` statement was `False`. Think of it as the program's way of saying, "If this condition is true then execute the if-block or else execute the else-block."

Remember to put the colon (the : sign) after the else keyword.

The Colon :

You may have noticed that we always place a colon at the end of `if`, `else`, `while`, and `def` statements. The colon marks the end of the statement, and tells us that the next line should be the beginning of a new block.

Code Explanation continued...

```
35. playAgain = 'yes'
```

This is the first line that is not a `def` statement or inside a def-block. This line is where our program really begins.

```
36. while playAgain == 'yes' or playAgain == 'y':
```

Here is the beginning of a `while` loop. We enter the loop if `playAgain` is equal to either `'yes'` or `'y'`. The first time we come to this while statement, we have just assigned the string value `'yes'` to the `playAgain` variable. That means this condition will be True.

```
38.     displayIntro()
```

Here we call the `displayIntro()` function. This isn't a Python function, it is our function that we defined earlier in our program. When this function is called, the program execution jumps to the first line in the `displayIntro()` function on line 5. When all the lines in the function are done, the execution jumps back down to the line after this one.

```
40.     caveNumber = chooseCave()
```

This line also calls a function that we created. Remember that the `chooseCave()` function lets the player type in the cave they choose to go into. When the return cave line in this function executes, the program execution jumps back down here, and the local variable `cave`'s value is the return value of this function. The return value is stored in a new variable named `caveNumber`. Then the execution moves to the next line.

```
42.     checkCave(caveNumber)
```

This line calls our `checkCave()` function with the argument of `caveNumber`'s value. Not only does execution jump to line 20, but the value stored in `caveNumber` is copied to the parameter `chosenCave` inside the `checkCave()` function. This is the function that will display either `'Gives you his treasure!'` or `'Gobbles you down in one bite!'`, depending on the cave the player chose to go in.

```
44.     print 'Do you want to play again? (yes or no)'
45.     playAgain = raw_input()
```

After the game has been played, the player is asked if they would like to play again. The variable `playAgain` stores the string that the user typed in. Then we reach the end of the while-block, so the program rechecks the `while` statement's condition: `while playAgain == 'yes' or playAgain == 'y'`

The difference is, now the value of `playAgain` is equal to whatever string the player typed in. If the player typed in the string `'yes'` or `'y'`, then we would enter the loop again at line 38.

If the player typed in `'no'` or `'n'` or something silly like `'Abraham Lincoln'`, then the `while` statement's condition would be `False`, and we would go to the next line after the while-block. But since there are no more lines after the while-block, the program terminates.

But remember, the string `'YES'` is different from the string `'yes'`. If the player typed in the string `'YES'`, then the `while` statement's condition would evaluate to `False` and the program would still terminate.

We've just completed our second game! In our Dragon Realm game, we used a lot of what we learned in the "Guess the Number" game and picked up a few new tricks as well. If you didn't understand some of the concepts in this program, then read the summary at the end of this chapter, or go over each line of the source code again, or try changing the source code and see how the program changes. In the next chapter we won't create a game, but a computer program that will create secret codes out of ordinary messages and also decode the secret code back to the original message.

Step by Step, One More Time

But first, let's trace the code one more time. A lot of new programming ideas were taught in this chapter. To help you understand everything, I will briefly go through the program just like the computer would, starting from the top.

```
1. import random
2. import time
```

We import the `random` and `time` modules, so that we can use the `random.randint()` and `time.sleep()` functions in our program.

```
4. def displayIntro():
```

This defines a new function. We do not execute the code inside this function block (because this function is being defined, not called). We just define it so that we can call this function later.

```
11. def chooseCave():
```

We define another function. Again, we skip past it for now.

```
19. def checkCave(chosenCave):
```

We define a third function. Again, we skip it for now because we are only defining the function, not calling it.

```
35. playAgain = 'yes'
```

The variable `playAgain` now has the string value of `'yes'`.

```
36. while playAgain == 'yes' or playAgain == 'y':
```

This is the start of a while-block. The value of `playAgain` is `'yes'` (we set it in the last line), so the condition evaluates to `True and False`, evaluates to `False`. But remember that `True` or `False` will evalute to `True`. So since the condition is `True`, we enter the `while` loop.

```
38.     displayIntro()
```

Now we are calling the `displayIntro()` function. You can tell this is a function call, and not defining a function because there is no `def` keyword in front of the function name. Now we jump back to the beginning of the `displayIntro()` function on line 4.

```
4. def displayIntro():
```

The program execution has jumped to line 4. We move down to the next line, which is line 5.

```
5.     print 'You are on a planet full of dragons. In front of you,'
6.     print 'you see two caves. In one cave, the dragon is friendly'
7.     print 'and will share his treasure with you. The other dragon'
8.     print 'is greedy and hungry, and will eat you on sight.'
9.     print
```

We print out the game introduction. This is the first text that the player sees, because this is the first time we have executed a `print` statement. The `print` statement without a string will just print a blank line. We know we have reached the end of this def-block, because line 11 does begin with less than four spaces. This means the execution jumps back down to line 38 (the line that sent us here).

```
38.     displayIntro()
```

We just made this call, so we move down to the next line.

```
40.     caveNumber = chooseCave()
```

We are going to assign the return value of the `chooseCave()` function to the variable `caveNumber`. In order to figure out what the return value is, we have to call the function. This moves the execution to line 11.

```
11. def chooseCave():
```

Here we are at line 11. This line doesn't do anything, it just marks the beginning of the `chooseCave` function. We move down to line 12.

```
12.     cave = ''
```

The value of the `cave` variable is now the blank string.

```
13.     while cave != '1' and cave != '2':
```

Here is the start of a while-block. To see if we enter the block or not, we check if the condition is true. `cave != '1'` evaluates to `True`, because the blank string is not equal to the string `'1'`. And `cave != '2'` also evaluates to `True`. So the condition evaluates to `True and True`. Both sides of the and operator must be `True` for the expression to evaluate to `True`, otherwise it will be `False`. `True and True` evaluates to `True`, so we do enter the `while` loop at line 14.

```
14.         print 'Which cave will you go into? (1 or 2)'
```

This line prints a question to the player. Move down to the next line.

```
15.         cave = raw_input()
```

The return value of the `raw_input()` function will be stored in the `cave` variable. The `raw_input()` question will let the player type in a string, and this string will be the return value. Let's say the player accidentally types in 3. The string value `'3'` will be the new value of the `cave`.

We have reached the end of the while-block (we know this because the next line (line 17) does not begin with 8 spaces). So execution jumps back to the top of the while-block at line 13.

```
13.     while cave != '1' and cave != '2':
```

Now we re-evaluate the while statement's condition. The value of cave this time is `'3'`. The `'3' != '1'` expression is `True`, and the `'3' != '2'` expression is `True`. And because `True and True` evaluates to `True`, we have to re-enter the while-block again.

Until the player types in the string `'1'` or `'2'`, this `while` statement's condition will be `True` and the program will keep asking the user for which `cave` they want to enter.

```
14.         print 'Which cave will you go into? (1 or 2)'
```

Again, we display a message that asks the player which `cave` they want to enter. Move down to the next line.

```
15.         cave = raw_input()
```

Let's say the player this time enters the string `'2'`. This string is stored in `cave`. We've reached the end of the while-block, so we jump back up to the start of the while-block to line 13 one more time.

```
13.     while cave != '1' and cave != '2':
```

Now we re-evaluate the `while` statement's condition. `cave` now equals `'2'`. The expression `cave != '1'` is `True`. The expression `cave != '2'` is `False`, because `cave` really does equal `'2'`. The expression evaluates to `True and False`. This expression evaluates to `False`. Because the `while` statement's condition is now `False`, we skip past the while-block to the next line after it, which is line 17.

```
17.     return cave
```

The `return` statement will return the value inside the `cave` variable, which is the string `'2'`. Execution goes back to where this function was called from, which was line 40.

```
40.     caveNumber = chooseCave()
```

Here we are back at line 40. But this time we know that the return value of this call to `chooseCave()` is the string `'2'`, so we store this string in the `caveNumber` variable. Move to the next line.

```
42.     checkCave(caveNumber)
```

Now we call the `checkCave()` function. This function has one parameter. We will pass the value inside `caveNumber` (the string `'2'`) as an argument for this parameter. Execution jumps up to line 19.

```
19. def checkCave(chosenCave):
```

Now we are at the top of the `checkCave()` function. Since the string `'2'` was passed as the argument for the first parameter, the variable `chosenCave` will be assigned the string `'2'`. When we jump back out of this function (and leave the function's scope), the value inside `chosenCave()` will be erased. Move down to the next line.

```
    print 'You approach the cave...'
```

We print a message to the player in this line.

```
    time.sleep(2)
```

We call the sleep() function (which is inside the time module). The sleep() function has one parameter, so we pass 2 as the argument for this parameter. The sleep() function pauses for however many seconds is given for the parameter. We make the program pause for two seconds, to add suspense for the player.

```
    print 'It is dark and spooky...'
    time.sleep(2)
```

We display another message to the player, and wait two seconds again.

```
    print 'A large dragon jumps out in front of you! He opens his jaws and...'
    print
    time.sleep(2)
```

We display another message telling the player the dragon has jumped out, followed by a blank line. We then pause again for two seconds, to let the player wonder what the dragon is going to do.

```
    friendlyCave = random.randint(1, 2)
```

The `friendlyCave` variable will be assigned the return value of the `random.randint()` function. The `randint()` function returns a random integer between the two numbers we pass to the two parameters. We want a random number between `1` and `2` (including `1` and `2`). Pretend that the integer `1` was returned by the function call. So `1` is stored in `friendlyCave`. Remember, this function call could have returned either a `1` or a `2`. We don't know which until we actually run the program, and it won't always return the same number each time we call the function. But we will just pretend that this time it returned a `1`.

```
30.     if chosenCave == str(friendlyCave):
```

This is an `if` statement. We must check if `'2'` (the string stored inside `chosenCave`) is equal to what str(friendlyCave) evaluates to. `str()` is a function that returns the string form of any integers passed to it. (Remember, the string value `'7'` is not equal to the integer value `7`.)

`friendlyCave` evaluates to the integer `1` that is stored inside of it. So the function call looks like `str(1)` and returns a value of `'1'`.

So now, to finish evaluating the `if` statement's condition, we see if `'2' == '1'`. This evaluates to `False`, because they are not the same. So execution skips past the if-block.

```
32.     else:
```

This is the first line after the if-block. It is the `else` keyword, which means that if the previous `if` statement's condition was `False`, we should execute the code inside the else-block. This means we move down one line to line 33, which is inside the else-block.

```
33.         print 'Gobbles you down in one bite!'
```

The player has been eaten. There are no more lines in the function, so execution goes back to line 42.

```
42.     checkCave(caveNumber)
```

We have just come back from the function call. Now we go to the next line down.

```
44.     print 'Do you want to play again? (yes or no)'
45.     playAgain = raw_input()
```

The player is asked if they want to play again. Let's pretend the player typed in the string `'no'`. This is the end of the while-block, so execution jumps back up to line 36.

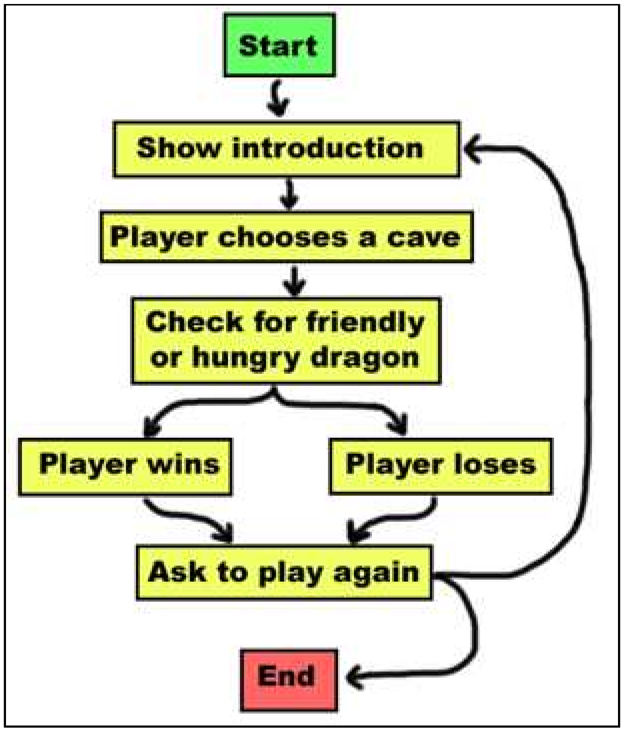
```
36. while playAgain == 'yes' or playAgain == 'y':
```

The `playAgain` variable contains the string `'no'`. That makes the condition evaluate to `False or False`, which evaluates to `False`. Since the condition is `False`, we skip past the while-block. But there are no more lines of code after the while-block, so the program terminates.

Designing the Program

Dragon Realm was a pretty simple game. The other games in this book will be a bit more complicated. It sometimes helps to write down everything you want your game or program to do before you start writing code. This is called "designing the program."

For example, it may help to draw a flow chart. A **flow chart** is a picture that shows every possible action that can happen in our game, and in what order. Normally we would create a flow chart before writing our program, so that we remember to write code for each thing that happens in the game. Here's a flow chart for Dragon Realm:



To see what happens in the game, put your finger on the "Start" box and follow one arrow from the box to another box. Your finger is kind of like the program execution. Your finger will trace out a path from box to box, until finally your finger lands on the "End" box. As you can see, when you get to the "Check for friendly or hungry dragon" box, the program could either go to the "Player wins" box or the "Player loses" box. Either way, both paths will end up at the "Ask to play again" box, and from there the program will either end or show the introduction to the player again.

A Web Page for Program Tracing

There is also a web page that lets you trace line by line through the Dragon Realm game. Go to the following web page to use it.

- Dragon Realm, trace 1 - http://pythonbook.coffeeghost.net/book1/traces/trace1DragonRealm.html

Things Covered In This Chapter:

- The `time` module.
- The `time.sleep()` function.
- The `return` keyword.
- Creating our own functions with the `def` keyword.
- The `and` and `or` and `not` boolean operators.
- Truth tables
- Variable scope (Global and Local)
- Parameters and Arguments

- Flow charts

Chapter 5 - Hangman

This game introduces many new concepts. But don't worry; we'll experiment with these programming concepts in the interactive shell first. Once you understand these concepts, it will be much easier to understand the game in this chapter: Hangman.

In case you don't know, Hangman is a game two people play with paper and pencil. One person thinks of a word, and then draws blanks for each letter in the secret word. The other person guesses letters that might be in the word. If they guess correctly, the first person writes the letter into the blank. If they guess wrong, the first person draws another body part of the hangman. If the second person can guess all the letters in the word before the hangman has completely been drawn, they win.

ASCII Art

The code for Hangman is about four times larger than our Dragon World game! But don't worry. Half of the lines of code aren't really code at all, but are strings that use keyboard characters to draw pictures. This type of graphics is called **ASCII art** (pronounced "ask-ee"), because keyboard characters (such as letters, numbers, and also all the other signs on the keyboard) are called ASCII characters. ASCII stands for American Standard Code for Information Interchange. Here are a couple cats done in ASCII art:

```
 ^___^
 |    )
 |. .  )
(  v    )
 \____  |
  |      \
  |       |
  |        \
  | |       |
  | | |      \
  | | |      |--.
  | | | _____/  |
 ((((()(_______/
                    _____________________
              _____/   xx      xxx       \_____
            _/xxx       xx      xxx      xxx   \____
         __/     xxx     xxx     xx       xxx       \
        /xxxxx     xx      xx     xx       xxx      x\_
       /xxxxxxxxx   xx     xx      xx       xx      xxx\_
      /      xxxxx    x     x       x        xx      xxx \
     /          xx  /\                       xx       xx  \
    /              /  \                      x        xx   \
    |   .=\        |   \                             xx    x\
    |   |  \       |    \                           xx   xxx|
    |   |   \      |     \____                          xx   \
```

```
         |   |    \____/           \                       xxx    |
         |   |                      \                             |
         |  /                        \                             \
          \/                          \                     |       \
          /                 ____/      |                   /        |
  _______|        \____                |                __/      xxx|
 /       |               ___   ___--------           __/           x|
/        |              |   |   ________          __/               |
|    oooo\    --------   \_/        __/       ___/            xxx  /
|oooo     \      _____/          __/_________/                 xx_/
\          \__                __/                          xx   /
 \            \______________/                              x__/
  \_                                                 x      /
    \_                                               xx  __/
      \_                                      xx      x_/
        \____                                  x _____/
             \__________________________________/
```

So this program's code is only about twice the size of Dragon World (if you don't count the pictures). Go ahead and type in this code into the file editor, and save the file as hangman.py. Then run the program by pressing F5. It might be a good idea to save the file every once in a while as you type it, so that if something happens to your computer or IDLE crashes, you won't lose everything you have typed.

Sample Run

```
H A N G M A N


    +------+
    |      |
    |      |
           |
           |
           |
           |
           |
           |
           |
           |
==============

Missed letters:
_ _ _ _ _
Guess a letter.
e
```

```
    +------+
    |      |
    |      |
           |
           |
           |
           |
           |
           |
           |
           |
==============

Missed letters:
_ _ _ e _
Guess a letter.
a


    +------+
    |      |
    |      |
    O      |
           |
           |
           |
           |
           |
           |
           |
==============

Missed letters: a
_ _ _ e _
Guess a letter.
u


    +------+
    |      |
    |      |
    O      |
    |      |
    |      |
    |      |
           |
```

```
           |
==============

Missed letters: a u
_ _ _ e _
Guess a letter.
r


    +------+
    |      |
    |      |
    O      |
    |      |
    |      |
    |      |
           |
           |
==============

Missed letters: a u
_ _ _ e r
Guess a letter.
i



    +------+
    |      |
    |      |
    O      |
   /|      |
  / |      |
    |      |
           |
           |
           |
           |
==============

Missed letters: a u i
_ _ _ e r
Guess a letter.
o
```

```
    +------+
    |      |
    |      |
    O      |
   /|      |
  / |      |
    |      |
           |
           |
           |
           |
==============

Missed letters: a u i
o _ _ e r
Guess a letter.
t
Yes! The secret word is "otter"! You have won!
Do you want to play again? (yes or no)
no
```

Source Code

hangman.py

```
import random

HANGMANPICS = ['''

    +------+
    |      |
    |      |
           |
           |
           |
           |
           |
           |
           |
           |
=============''', '''

    +------+
    |      |
    |      |
```

```
    O      |
           |
           |
           |
           |
           |
           |
           |
=============''', '''

    +------+
    |      |
    |      |
    O      |
    |      |
    |      |
    |      |
           |
           |
=============''', '''


    +------+
    |      |
    |      |
    O      |
   /|      |
  / |      |
    |      |
           |
           |
           |
           |
=============''', '''


    +------+
    |      |
    |      |
    O      |
   /|\     |
  / | \    |
    |      |
           |
           |
           |
           |
```

```
==============''', '''


    +------+
    |      |
    |      |
    O      |
   /|\     |
  / | \    |
    |      |
   /       |
  /        |
           |
           |
==============''', '''


    +------+
    |      |
    |      |
    O      |
   /|\     |
  / | \    |
    |      |
   / \     |
  /   \    |
           |
           |
==============''']

words = 'ant baboon badger bat bear beaver beetle bird camel cat clam cobra cougar coyote crab crane crow deer dog donkey duck eagle ferret fish fox frog goat goose hawk iguana jackal koala leech lemur lion lizard llama mite mole monkey moose moth mouse mule newt otter owl oyster panda parrot pigeon python quail rabbit ram rat raven rhino salmon seal shark sheep skunk sloth slug snail snake spider squid stork swan tick tiger toad trout turkey turtle wasp weasel whale wolf wombat worm zebra'.split()

def getRandomWord(wordList):
    # This function returns a random string from the passed list of strings.
    wordIndex = random.randint(0, len(wordList) - 1)
    return wordList[wordIndex]

```

```
def displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord):
    print HANGMANPICS[len(missedLetters)]
    print

    print 'Missed letters:',
    for letter in missedLetters:
        print letter,
    print

    blanks = '_' * len(secretWord)

    for i in range(len(secretWord)): # replace blanks with correctly guessed letters
        if secretWord[i] in correctLetters:
            blanks = blanks[:i] + secretWord[i] + blanks[i+1:]

    for letter in blanks: # show the secret word with spaces in between each letter
        print letter,
    print

def getGuess(alreadyGuessed):
    # Returns the letter the player entered. This function makes sure the player entered a single letter, and not something else.
    while True:
        print 'Guess a letter.'
        guess = raw_input()
        guess = guess.lower()
        if len(guess) != 1:
            print 'Please enter a single letter.'
        elif guess in alreadyGuessed:
            print 'You have already guessed that letter. Choose again.'
        elif guess not in 'abcdefghijklmnopqrstuvwxyz':
            print 'Please enter a LETTER.'
        else:
            return guess

def playAgain():
    # This function returns True if the player wants to play again, otherwise it returns False.
    print 'Do you want to play again? (yes or no)'
    return raw_input().lower().startswith('y')

```

```

print 'H A N G M A N'
missedLetters = ''
correctLetters = ''
secretWord = getRandomWord(words)
gameIsDone = False

while True:
    displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)

    # Let the player type in a letter.
    guess = getGuess(missedLetters + correctLetters)

    if guess in secretWord:
        correctLetters = correctLetters + guess

        # Check if the player has won
        foundAllLetters = True
        for i in range(len(secretWord)):
            if secretWord[i] not in correctLetters:
                foundAllLetters = False
                break
        if foundAllLetters:
            print 'Yes! The secret word is "' + secretWord + '"! You have won!'
            gameIsDone = True
    else:
        missedLetters = missedLetters + guess

        # Check if player has guessed too many times and lost
        if len(missedLetters) == len(HANGMANPICS) - 1:
            displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)
            print 'You have run out of guesses!\nAfter ' + str(len(missedLetters)) + ' missed guesses and ' + str(len(correctLetters)) + ' correct guesses, the word was "' + secretWord + '"'
            gameIsDone = True

    # Ask the player if they want to play again (but only if the game is done).
    if gameIsDone:
        if playAgain():
            missedLetters = ''
            correctLetters = ''
```

```
            gameIsDone = False
            secretWord = getRandomWord(words)
        else:
            break

```

After typing in the source code (don't forget to save!) you can run this game by pressing F5. If any errors come up, be sure you typed the source code in *exactly* as it appears here. Remember that the indentation is important, and that lines will have zero, four, eight, or even twelve spaces in front of them.

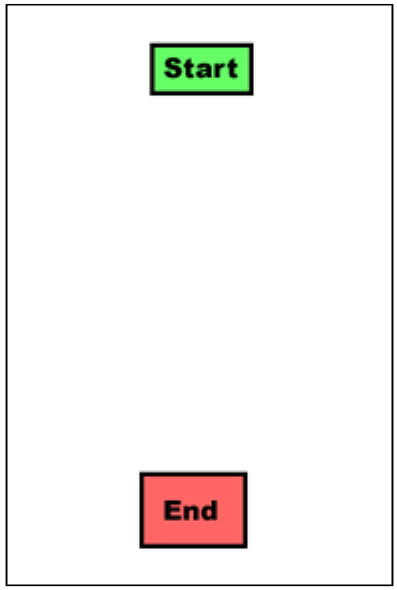
Designing the Program

This game is a bit more complicated, so it will help if we take a moment to think about how we will put the program together. We will create a flow chart (like the flow chart at the end of the Dragon Realm chapter) to think about what this program will do. Of course, we don't *have* to write out a flow chart. We could just start writing code. But many times when we are writing code, we will think of new things to add or other events in the program that we didn't think of. We may have to end up changing the code we have already written, or deleting a lot of the code. That would be a waste of effort. We can save a lot of time if we think about the program before writing it.

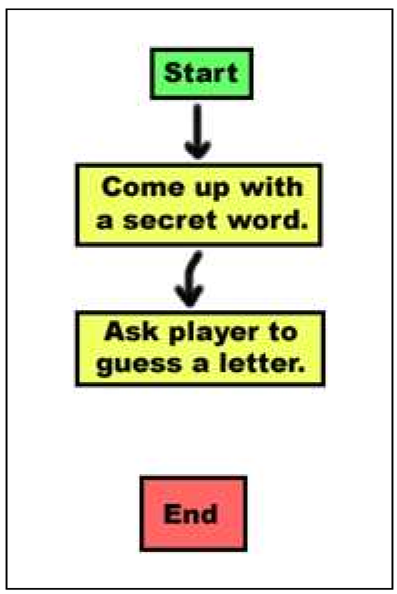
This flow chart is provided as an example for what flow charts look like and how to make them. Because you only have to copy the source code from this book, you don't need to draw a flow chart before writing code. But when you make your own games, a flow chart can be very handy.

This flow chart will also help you learn how to design games yourself, instead of just copying the source code from this book. Your flow chart doesn't have to look exactly like this one. You may have extra boxes or fewer boxes. But as long as *you* understand the flow chart you made, it will be helpful when you start coding.

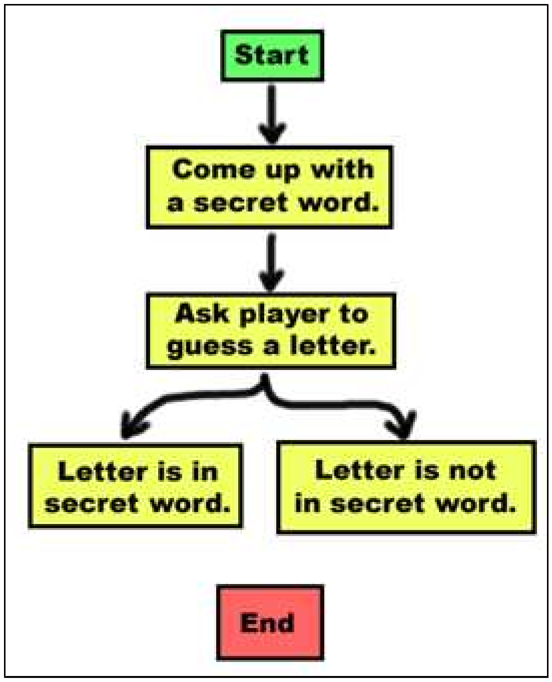
First we'll start with a flow chart that only has a "Start" and an "End" box:



Now let's think about what happens when the play Hangman. Well, there is a secret word (the computer will think of this) and then the other person guesses letters (the player will do this) so let's add boxes for those:



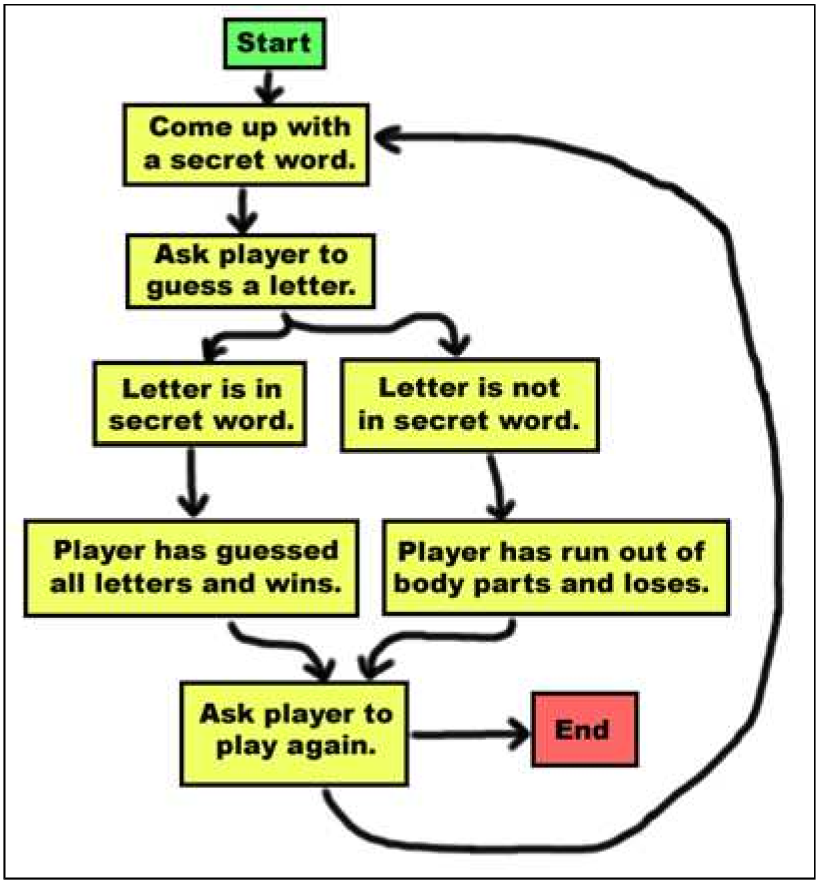
The game doesn't end after the player guesses a letter. The game should check if the letter is in the secret word or not. The letter either will be there or it won't be, so we should put *two* new boxes in:



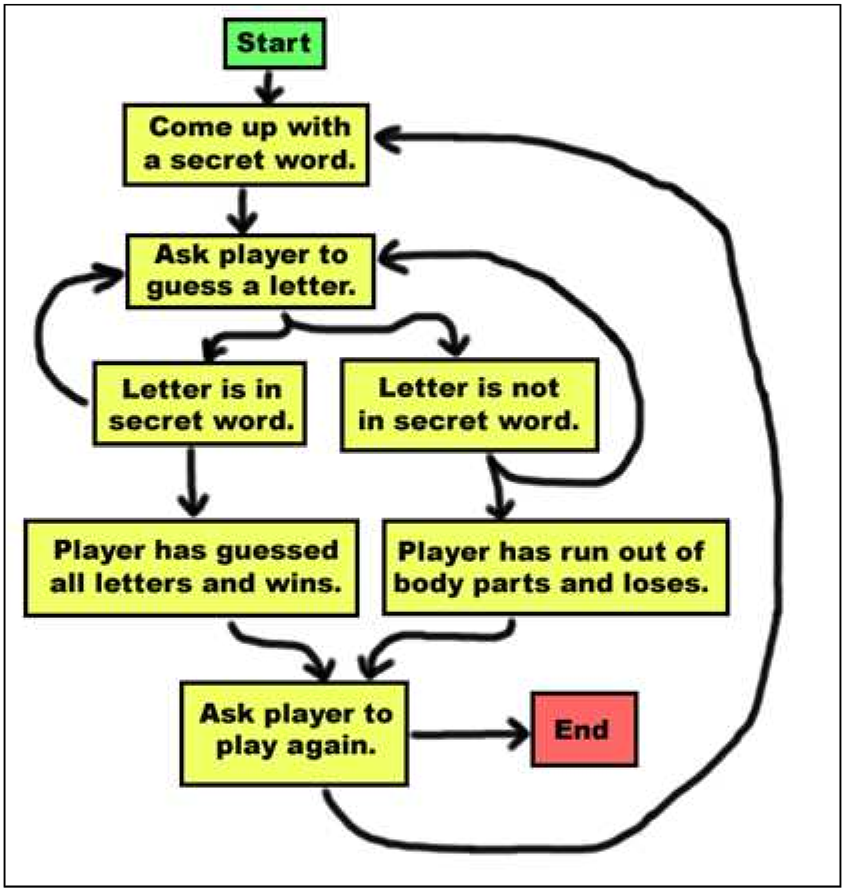
If the letter was in the secret word, we should also check if the player has won. And if the letter was not in the secret word, another body part gets added and the player might have lost. We can add boxes for those cases too. We don't need an arrow from the "Letter is in secret word." box to the "Player has run out of body parts and loses." box because if you think about it, you cannot possibly lose as long as you are guessing correct letters. Also, you cannot possibly win if you are guessing wrong letters. That is why we don't have those arrows. The flow chart now looks like:



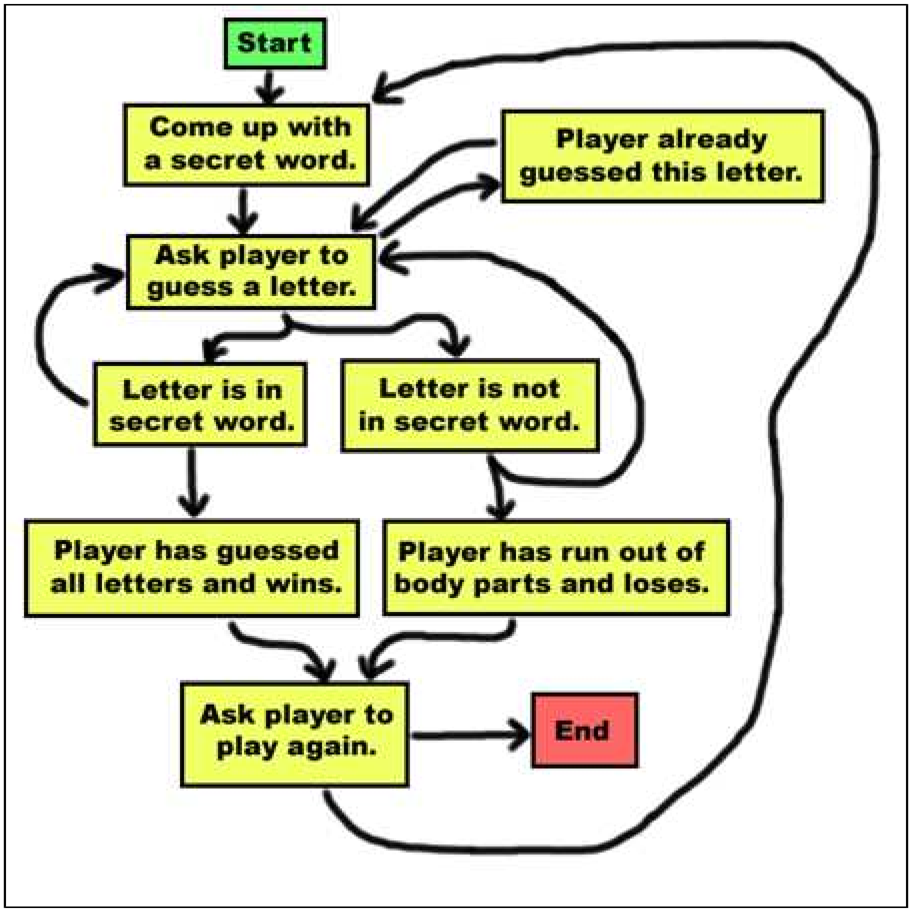
After the player has won or lost, we will ask the player if they want to play again with a new secret word. If the player doesn't want to play again, the program will terminate. Otherwise, we will think of a new secret word.



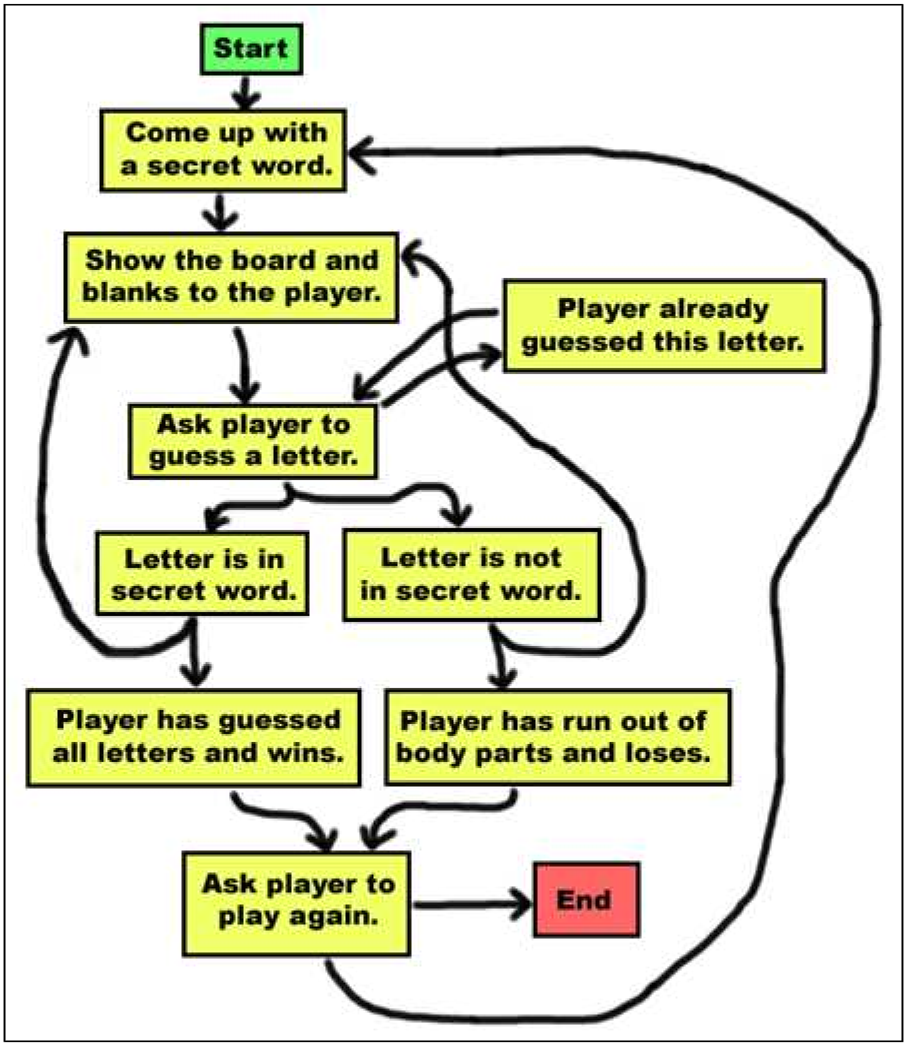
This flow chart looks like it is finished, but is there something we are forgetting? Oh yes! The player doesn't guess a letter just once. The player will have to keep guessing letters over and over until they win or lose. We should draw two new arrows so the flow chart shows this.



What else are we forgetting? What if when the player guesses a letter, they guess a letter they have guessed before. The player should not win or lose in this case, but should be allowed to guess a different letter instead:



Wait a second. How can the player figure out how well or how bad they're doing in this game? We need to remember to show them the hangman board and also the secret word (with the unguessed letters blanked out). Then the player will be able to see how many body parts of the hangman there are, or how much of the secret word they have guessed so far. This should happen each time the player guesses. We can add this box in between the "Come up with a secret word." box and the "Ask player to guess a letter." box:



That looks good! We can always look at this flow chart while we are coding to remind ourselves of everything we want this program to do. The flow chart is kind of like a cake recipe or blueprints for a house. We could just start baking a cake or building a house, but without the plans we may forget to do a step. You won't really need this flow chart because you will just copy the source code given here. But when you design your own games, a flow chart can help you remember everything you need to code.

Code Explanation

```
1. import random
```

The Hangman program is going to randomly select a secret word from a list of secret words. This means we will need the `random` module imported.

```
 3. HANGMANPICS = ['''
 4.
 5.     +------+
 6.     |      |
 7.     |      |
 8.            |
 9.            |
10.            |
11.            |
12.            |
13.            |
14.            |
15.            |
16. =============''', '''

 ...the rest of the code is too big to show here...
```

This "line" of code a simple variable assignment, but it actually stretches over several real lines in the source code. The actual "line" doesn't end until line 96. To help you understand what this code means, you should learn about multi-line strings and lists:

Multi-line Strings

Ordinarily when you write strings in your source code, the string has to be on one line. However, if you use three single-quotes instead of one single-quote to begin and end the string, the string can be on several lines:

```
>>> fizz = '''Dear Alice,
I will return home at the end of the month. I will see you then.
Your friend,
Bob'''
>>> print fizz
Dear Alice,
I will return home at the end of the month. I will see you then.
Your friend,
Bob
>>> 
```

If we didn't have multi-line strings, we would have to use the `\n` escape character to represent the new lines. But that can make the string hard to read in the source code:

```
>>> fizz = 'Dear Alice,\nI will return home at the end of the month. I will see
you then.\nYour friend,\nBob'
>>> print fizz
Dear Alice,
I will return home at the end of the month. I will see you then.
Your friend,
Bob
>>>
```

Multi-line strings do not have to keep the same indentation to remain in the same block. Within the multi-line string, Python ignores the indentation rules it normally has for where blocks end.

```
def writeLetter():
    # inside the def-block
    print '''Dear Alice,
How are you? Write back to me soon.

Sincerely,
  Bob''' # end of the multi-line string and print statement
    print 'P.S. I miss you.' # still inside the def-block

writeLetter() # This is the first line outside the def-
block.
```

Constant Variables

You may have noticed that `HANGMANPICS`'s name is in all capitals. This is the programming convention for constant variables. **Constants** are variables whose values do not change throughout the program. Although we can change `HANGMANPICS` just like any other variable, the all-caps reminds the programmer to not write code that does so.

Constant variables are helpful for providing descriptions for values that have a special meaning. Since the multi-string value never changes, there is no reason we couldn't copy this multi-line string each time we needed that value. The `HANGMANPICS` variable never varies. But it is much shorter to type `HANGMANPICS` than it is to type that large multi-line string.

Also, there are cases where typing the value by itself may not be obvious. If we set a variable `eggs = 72`, we may forget why we were setting that variable to the integer `72`. But if we define a constant variable `DOZEN = 12`, then we could set `eggs = DOZEN * 6` and by just looking at the code know that the `eggs` variable was set to six dozen.

Like all conventions, we don't *have* to use constant variables, or even put the names of constant variables in all capitals. But doing it this way makes it easier for other programmers to understand how these variables are used. (It even can help you if you are looking at code you wrote a long time ago.)

Lists

I will now tell you about a new data type called a **list**. A list value can contain several other values in it. Try typing this into the shell: `['apples', 'oranges', 'HELLO WORLD']`. This is a list value that contains three string values. Just like any other value, you can store this list in a variable. Try typing `spam = ['apples', 'oranges', 'HELLO WORLD']`, and then type `spam` to view the contents of `spam`.

```
>>> spam = ['apples', 'oranges', 'HELLO WORLD']
>>> spam
['apples', 'oranges', 'HELLO WORLD']
>>>
```

Lists are a good way to store several different values into one variable. The individual values inside of a list are also called **items**. Try typing: `animals = ['aardvark', 'anteater', 'antelope', 'albert']` to store various strings into the variable `animals`. The square brackets can also be used to get an item from a list. Try typing `animals[0]`, or `animals[1]`, or `animals[2]`, or `animals[3]` into the shell to see what they evaluate to.

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[0]
'aardvark'
>>> animals[1]
'anteater'
>>> animals[2]
'antelope'
>>> animals[3]
'albert'
>>>
```

The number between the square brackets is the **index**. In Python, the first index is the number 0 instead of the number 1. So the first item in the list is at index 0, the second item is at index 1, the third item is at index 2, and so on. Lists are very good when we have to store lots and lots of values, but we don't want variables for each one. Otherwise we would have something like this:

```
>>> animal1 = 'aardvark'
>>> animal2 = 'anteater'
>>> animal3 = 'antelope'
>>> animal4 = 'albert'
>>>
```

This makes working with all the strings as a group very hard, especially if you have hundreds or thousands (or even millions) of different strings that you want stored in a list. Using the square brackets, you can treat items in the list just like any other value. Try typing `animals[0] + animals[2]` into the shell:

```
>>> animals[0] + animals[2]
'aardvarkantelope'
>>>
```

Because `animals[0]` evaluates to the string `'aardvark'` and `animals[2]` evaluates to the string `'antelope'`, then the expression `animals[0] + animals[2]` is the same as `'aardvark' + 'antelope'`. This string concatenation evaluates to `'aardvarkantelope'`.

What happens if we enter an index that is larger than the list's largest index? Try typing `animals[4]` or `animals[99]` into the shell:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[4]

Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    animals[4]
IndexError: list index out of range
>>> animals[99]

Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    animals[99]
IndexError: list index out of range
>>>
```

If you try accessing an index that is too large, you will get an **index error**.

Changing the Values of List Items with Index Assignment

You can also use the square brackets to change the value of an item in a list. Try typing `animals[1] = 'ANTEATER'`, then type `animals` to view the list.

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[1] = 'ANTEATER'
>>> animals
['aardvark', 'ANTEATER', 'antelope', 'albert']
>>>
```

The second item in the `animals` list has been overwritten with a new string.

List Concatenation

You can join lists together into one list with the `+` operator, just like you can join strings. When joining lists, this is known as **list concatenation**. Try typing `[1, 2, 3, 4] + ['apples', 'oranges'] + ['Alice', 'Bob']` into the shell:

```
>>> [1, 2, 3, 4] + ['apples', 'oranges'] + ['Alice', 'Bob']
[1, 2, 3, 4, 'apples', 'oranges', 'Alice', 'Bob']
>>>
```

Notice that lists do not have to store values of the same data types. The example above has a list with both integers and strings in it.

The `in` Operator

The `in` operator makes it easy to see if a value is inside a list or not. Expressions that use the `in` operator return a boolean value: `True` if the value is in the list and `False` if the value is not in the list. Try typing `'antelope' in animals` into the shell:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> 'antelope' in animals
True
>>> 
```

The expression `'antelope' in animals` returns `True` because the string `'antelope'` can be found in the list, `animals`. (It is located at index 2.)

But if we type the expression `'ant' in animals`, this will return `False` because the string `'ant'` does not exist in the list. We can try the expression `'ant' in ['beetle', 'wasp', 'ant']`, and see that it will return `True`.

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> 'antelope' in animals
True
>>> 'ant' in animals
False
>>> 'ant' in ['beetle', 'wasp', 'ant']
True
>>>
```

The `in` operator also works for strings as well as lists. You can check if one string exists in another the same way you can check if a value exists in a list. Try typing `'hello' in 'Alice said hello to Bob.'` into the shell. This expression will evaluate to `True`.

```
>>> 'hello' in 'Alice said hello to Bob.'
True
>>> 
```

Removing Items from Lists with `del` Statements

You can remove items from a list with a `del` statement. ("del" is short for "delete.") Try creating a list of numbers by typing: `spam = [2, 4, 6, 8, 10]` and then `del spam[1]`. Type `spam` to view the list's contents:

```
>>> spam = [2, 4, 6, 8, 10]
>>> del spam[1]
>>> spam
[2, 6, 8, 10]
>>>
```

Notice that when you deleted the item at index 1, the item that used to be at index 2 became the new index 1. The item that used to be at index 3 moved to be the new index 2. Everything above the item that we deleted moved down one index. We can type `del spam[1]` again and again to keep deleting items from the list:

```
>>> spam = [2, 4, 6, 8, 10]
>>> del spam[1]
>>> spam
[2, 6, 8, 10]
>>> del spam[1]
>>> spam
[2, 8, 10]
>>> del spam[1]
>>> spam
[2, 10]
>>>
```

Lists of Lists

Lists are a data type that can contain other values as items in the list. But these items can also be other lists. Let's say you have a list of groceries, a list of chores, and a list of your favorite pies. You can put all three of these lists into another list. Try typing this into the shell:

```
groceries = ['eggs', 'milk', 'soup', 'apples', 'bread']
chores = ['clean', 'mow the lawn', 'go grocery shopping']
favoritePies = ['apple', 'frumbleberry']
listOfLists = [groceries, chores, favoritePies]
listOfLists
```

```
>>> groceries = ['eggs', 'milk', 'soup', 'apples', 'bread']
>>> chores = ['clean', 'mow the lawn', 'go grocery shopping']
>>> favoritePies = ['apple', 'frumbleberry']
>>> listOfLists = [groceries, chores, favoritePies]
>>> listOfLists
[['eggs', 'milk', 'soup', 'apples', 'bread'], ['clean', 'mow the lawn', 'go groc
ery shopping'], ['apple', 'frumbleberry']]
>>>
```

You could also type the following and get the same values for all four variables:

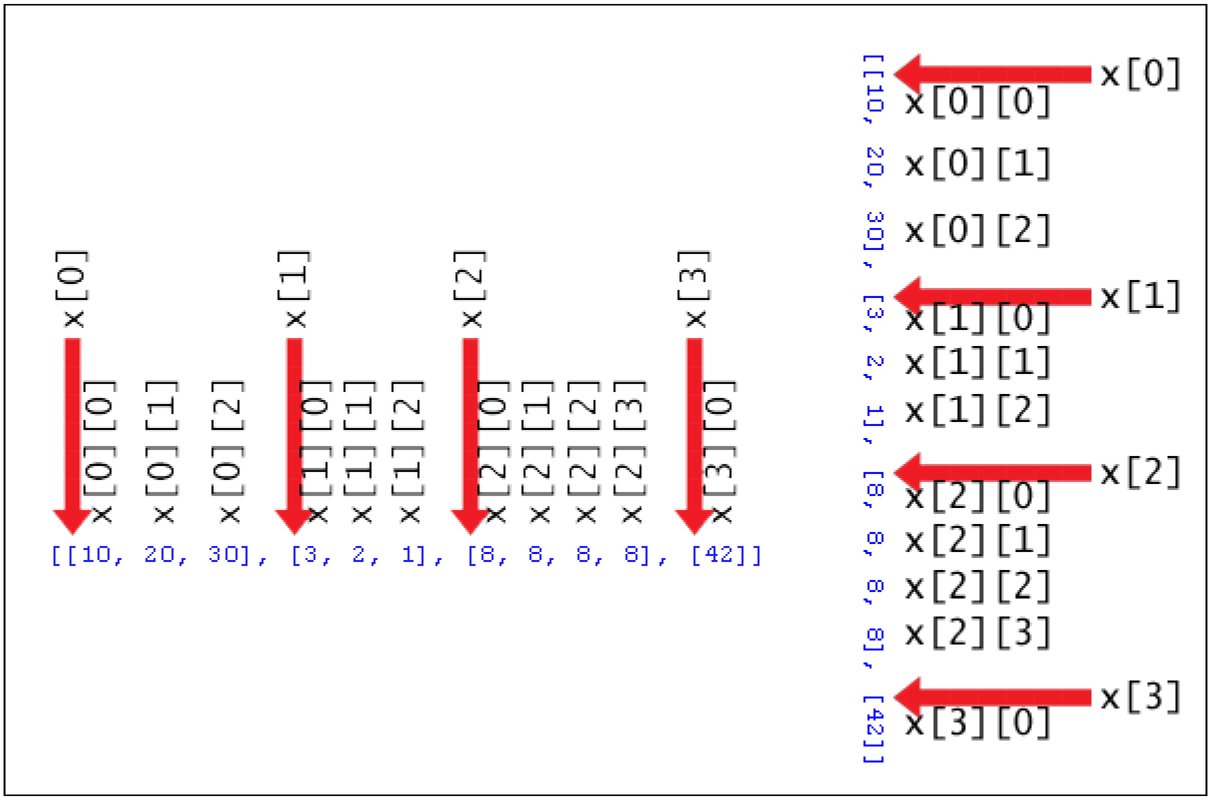
```
listOfLists = [['eggs', 'milk', 'soup', 'apples', 'bread'], ['clean',
```

```
  'mow the lawn', 'go grocery shopping'], ['apple', 'frumbleberry']]
groceries = listOfLists[0]
chores = listOfLists[1]
favoritePies = listOfLists[2]
listOfLists
```

```
>>> listOfLists = [['eggs', 'milk', 'soup', 'apples', 'bread'], ['clean', 'mow t
he lawn', 'go grocery shopping'], ['apple', 'frumbleberry']]
>>> groceries = listOfLists[0]
>>> chores = listOfLists[1]
>>> favoritePies = listOfLists[2]
>>> listOfLists
[['eggs', 'milk', 'soup', 'apples', 'bread'], ['clean', 'mow the lawn', 'go groc
ery shopping'], ['apple', 'frumbleberry']]
>>> groceries
['eggs', 'milk', 'soup', 'apples', 'bread']
>>> chores
['clean', 'mow the lawn', 'go grocery shopping']
>>> favoritePies
['apple', 'frumbleberry']
>>> 
```

To get an item inside the list of lists, you would use *two* sets of square brackets like this: `listOfLists[1][2]` which would evaluate to the string `'go grocery shopping'`. This is because `listOfLists[1]` evaluates to the list `['clean', 'mow the lawn', 'go grocery shopping'][2]`. That finally evaluates to `'go grocery shopping'`.

Here is another example of a list of lists, along with some of the indexes that point to the items in the list of lists named `x`. The red arrows point to indexes of the inner lists themselves. The image is also flipped on its side to make it easier to read:



Code Explanation continued...

```
 3. HANGMANPICS = ['''
 4.
 5.     +------+
 6.     |      |
 7.     |      |
 8.            |
 9.            |
10.            |
11.            |
12.            |
13.            |
14.            |
15.            |
16. ==============''', '''

 ...the rest of the code is too big to show here...
```

If you look from line 3 to line 96 in the code, you will see that the value we are assigning to the

variable HANGMANPICS is a list of multi-line strings. Each multi-line string in this list will be the picture (in ASCII art) of the hangman board. The string at HANGMANPICS[0] is the hangman's noose with no body parts. The string at HANGMANPICS[1] has just the head, HANGMANPICS[2] has the head and body, and so on.

```
98. words = 'ant baboon badger bat bear beaver beetle bird
    camel cat clam cobra cougar coyote crab crane crow
    deer dog donkey duck eagle ferret fish fox frog goat
    goose hawk iguana jackal koala leech lemur lion lizard
    llama mite mole monkey moose moth mouse mule newt
    otter owl oyster panda parrot pigeon python quail
    rabbit ram rat raven rhino salmon seal shark sheep
    skunk sloth slug snail snake spider squid stork swan
    tick tiger toad trout turkey turtle wasp weasel whale
    wolf wombat worm zebra'.split()
```

Line 98 assigns a list to the variable words. This will be the list of all possible secret words in this game. The secret word will be selected from this list. All of the possible secret words are some kind of animal (so the player has some idea what the word is).

But the value being assigned to words doesn't look like a list. It does not have the [and] square brackets. But there is a special kind of function call at the end of the long string, .split(). This is a method on the string, and it will evaluate to a list which is then stored in words. Read on to find out what methods are.

Methods

Methods are functions that are attached with a certain value. For example, the strings have a lower() method. You cannot just call the lower() by itself. You must attach the method call to a specific string. Try typing 'Hello world!'.lower() into the interactive shell:

```
>>> 'Hello world!'.lower()
'hello world!'
>>>
```

The lower() method returns the lowercase version of the string it is attached to. There is also an upper() method for strings. Try tying 'Hello world'.upper() into the shell:

```
>>> 'Hello world!'.lower()
'hello world!'
>>> 'Hello world!'.upper()
'HELLO WORLD!'
>>>
```

Because the `upper()` method returns a string, you can call a method on *that* string as well. Try typing `'Hello world!'.upper().lower()` into the shell:

```
>>> 'Hello world!'.lower()
'hello world!'
>>> 'Hello world!'.upper()
'HELLO WORLD!'
>>> 'Hello world!'.upper().lower()
'hello world!'
>>>
```

`'Hello world!'.upper()` evaluates to the string `'HELLO WORLD!'`, and then we call *that* string's `lower()` method. This returns the string `'hello world!'`, which is the final value in the evaluation. The order is important. `'Hello world!'.lower().upper()` is not the same as `'Hello world!'.upper().lower()`:

```
>>> 'Hello world!'.lower().upper()
'HELLO WORLD!'
>>> 'Hello world!'.upper().lower()
'hello world!'
>>>
```

Remember, if a string is stored in a variable, you can call a string method on that variable. Look at this example:

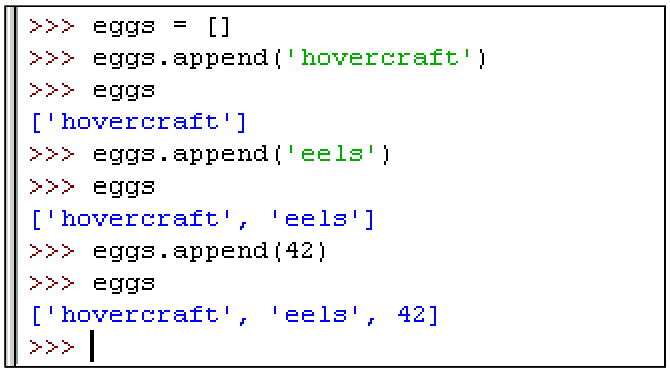
```
>>> fizz = 'Hello world'
>>> fizz.upper()
'HELLO WORLD'
>>>
```

The list data type also has methods. The `reverse()` method will reverse the order of the items in the list. Try typing `spam = [1, 2, 3, 4, 5, 6, 'meow', 'woof']` and then `spam.reverse()` (to reverse the list). Then type `spam` to view the contents of the `spam` variable.

```
>>> spam = [1, 2, 3, 4, 5, 6, 'meow', 'woof']
>>> spam.reverse()
>>> spam
['woof', 'meow', 6, 5, 4, 3, 2, 1]
>>>
```

The most common list method you will use is `append()`. This method will add the value you pass as an argument to the end of the list. Try typing the following into the shell:

```
eggs = []
eggs.append('hovercraft')
eggs
eggs.append('eels')
eggs
eggs.append(42)
eggs
```



```
>>> eggs = []
>>> eggs.append('hovercraft')
>>> eggs
['hovercraft']
>>> eggs.append('eels')
>>> eggs
['hovercraft', 'eels']
>>> eggs.append(42)
>>> eggs
['hovercraft', 'eels', 42]
>>>
```

While strings and lists have methods, integers do not happen to have any methods.

You may be wondering why Python has methods anyway, since they do the same thing as functions. Attaching functions to values (which is what methods are) becomes a lot more useful in object-oriented programming (OOP). Strings and lists are also known as a special type of data type called objects. But object-oriented programming is a bit advanced for this book, and you don't need to know OOP to make these games. You only need to know about string methods and the `append()` list method.

Code Explanation continued...

```
words = 'ant baboon badger bat bear beaver beetle bird
camel cat clam cobra cougar coyote crab crane crow
deer dog donkey duck eagle ferret fish fox frog goat
goose hawk iguana jackal koala leech lemur lion lizard
llama mite mole monkey moose moth mouse mule newt
otter owl oyster panda parrot pigeon python quail
rabbit ram rat raven rhino salmon seal shark sheep
skunk sloth slug snail snake spider squid stork swan
tick tiger toad trout turkey turtle wasp weasel whale
wolf wombat worm zebra'.split()
```

As you can see, this line is just one very, very long string that has the `split()` method called on it. The `split()` method will return a list made up of the words in the string that are separated by a space.

(The string is split up into a list of items.) The reason we do it this way instead of just writing out the list is that it is easier to type as one long string. Otherwise you would have to type: `['ant', 'baboon', 'badger',`... with all the quotes and commas. The `words` list will contain the possible secret words that can show up in the game. You can add or remove your own words to this string later if you want to change the words used in the Hangman game.

For an example of how the `split()` string method works, try typing `'My very energetic mother just served us nine pies'.split()` into the shell:

```
>>> 'My very energetic mother just served us nine pies'.split()
['My', 'very', 'energetic', 'mother', 'just', 'served', 'us', 'nine', 'pies']
>>> 
```

The result is a list of nine strings, one string for each of the words in the original string. The spaces are dropped from the items in the list.

```
100. def getRandomWord(wordList):
101.     # This function returns a random string from the
     passed list of strings.
102.     wordIndex = random.randint(0, len(wordList) - 1)
103.     return wordList[wordIndex]
```

Starting on line 100, we define a new function called `getRandomWord()` which has a single parameter named `wordList`. We will call this function when we want to pick a secret word from a list of secret words. This function makes use of a new Python function named `len()`, which I will explain first.

The len() Function

The `len()` function ("len" is short for "length") takes a list as a parameter and returns the integer of how many items are in a list. Try typing `len(animals)` into the shell:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> len(animals)
4
>>> 
```

The integer value returned by `len()` is like any other integer value:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> len(animals)
4
>>> people = ['Alice', 'Bob']
>>> len(people)
2
>>> len(animals) + len(people)
6
>>>
```

The square brackets by themselves are also a list value known as the **empty list**. If you pass the empty list to the `len()` function, it returns the integer `0`, because there are zero items in that list:

```
>>> len([])
0
>>> spam = []
>>> len(spam)
0
>>>
```

Code Explanation continued...

```
100. def getRandomWord(wordList):
101.     # This function returns a random string from the passed list of strings.
102.     wordIndex = random.randint(0, len(wordList) - 1)
103.     return wordList[wordIndex]
```

The function `getRandomWord()` is passed a list of strings as the argument for the `wordList` parameter. On line 102, we will store a random index in this list in the `wordIndex` variable. We do this by calling `randint()` with two arguments. Remember that arguments in a function call are separated by commas, so the first argument is `0` and the second argument is `len(wordList) - 1`. The second argument is an expression that is first evaluated. `len(wordList)` will return the integer size of the list passed to `getRandomWord()`, minus one.

For example, if we passed `['apple', 'orange', grape']` as an argument to `getRandomWord()`, then `len(wordList)` would return the integer `3` and the expression `3 - 1` would evaluate to the integer `2`.

That means that `wordIndex` would contain the return value of `randint(0, 2)`, which means `wordIndex` would equal `0`, `1`, or `2`. On line 103, we would return the element in `wordList` at the integer index stored in `wordIndex`.

Let's pretend we did send `['apple', 'orange', grape']` as the argument to `getRandomWord()` and that `randint(0, 2)` returned the integer `2`. That would mean that line 103 would become `return wordList[2]`, which would evaluate to `return 'grape'`. This is how the `getRandomWord()` returns a random string in the `wordList` list.

But remember, we can pass any list of strings to `getRandomWord()`. This function will be very useful to our Hangman game when we call it.

```
def displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord):
    print HANGMANPICS[len(missedLetters)]
    print
```

This code defines a new function named `displayBoard()`. This function has four parameters. This function will implement the code for the "Show the board and blanks to the player" box in our flow chart. Here is what each parameter means:

- `HANGMANPICS` - This is a list of multi-line strings that will display the board as ASCII art. We will always pass the global `HANGMANPICS` variable as the argument for this parameter.
- `missedLetters` - This is a string made up of the letters the player has guessed that are not in the secret word.
- `correctLetters` - This is a string made up of the letters the player has guessed that are in the secret word.
- `secretWord` - This string is the secret word.

The first `print` statement will display the board. `HANGMANPICS` will be a list of strings for each possible board. `HANGMANPICS[0]` shows an empty gallows, `HANGMANPICS[1]` shows the head (this happens when the player misses one letter), `HANGMANPICS[2]` shows a head and body (this happens when the player misses two letters), and so on until `HANGMANPICS[6]` when the full hangman is shown and the player loses.

The number of letters in `missedLetters` will tell us how many missed guesses the player has made. We can call `len(missedLetters)` to find out this number. This number can also be used as the index to the `HANGMANPICS` list for the specific string we want to print. So, if `missedLetters` is `'aetr'` then `len('aetr')` will return `4` and we will display the string `HANGMANPICS[4]`. This is what `HANGMANPICS[len(missedLetters)]` evaluates to. This line shows the correct hangman board to the player.

```
    print 'Missed letters:',
```

```
110.     for letter in missedLetters:
111.         print letter,
112.     print
```

Line 110 is a new type of loop, called a `for` loop. They are kind of like `while` loops. Line 111 is the entire body of the `for` loop. The `range()` function is often used with `for` loops. I will explain both in the next two sections.

The `range()` Function

The `range()` function is easy. You can call it with either one or two integer arguments. When called with one argument, `range()` will return a list of integers from 0 up to (but not including) the argument. Try typing `range(10)` into the shell:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

It's very easy to generate huge lists with the `range()` function. Try typing in `range(10000)` into the shell:

```
 9905, 9906, 9907, 9908, 9909, 9910, 9911, 9912, 9913, 9914, 9915, 9916, 9917, 9
918, 9919, 9920, 9921, 9922, 9923, 9924, 9925, 9926, 9927, 9928, 9929, 9930, 993
1, 9932, 9933, 9934, 9935, 9936, 9937, 9938, 9939, 9940, 9941, 9942, 9943, 9944,
 9945, 9946, 9947, 9948, 9949, 9950, 9951, 9952, 9953, 9954, 9955, 9956, 9957, 9
958, 9959, 9960, 9961, 9962, 9963, 9964, 9965, 9966, 9967, 9968, 9969, 9970, 997
1, 9972, 9973, 9974, 9975, 9976, 9977, 9978, 9979, 9980, 9981, 9982, 9983, 9984,
 9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9
998, 9999]
>>>
                                                                     Ln: 143 Co
```

The list is so huge, that it won't even all fit onto the screen. But we can save the list into the variable just like any other list by typing `spam = range(10000)`

```
 9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9
998, 9999]
>>> spam = range(10000)
>>>
                                                                     Ln: 144 Co
```

If you pass two arguments to `range()`, the list of integers it returns is from the first argument up to (but not including) the second argument. Try typing `range(10, 20)` into the shell:

```
>>> range(10, 20)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>>
```

The `range()` is a very useful function, because we often use it in `for` loops (which are much like the `while` loops we have already seen).

`for` Loops

The `for` loop is very good at looping over a list of values. This is different from the `while` loop, which loops as long as a certain condition is true. A `for` statement begins with the `for` keyword, followed by a variable, followed by the `in` keyword, followed by a sequence (such as a list or string) and then a colon. Each time the program execution goes through the loop (that is, on each **iteration** through the loop) the variable in the `for` statement takes on the value of the next item in the list.

For example, you just learned that the `range()` function will return a list of integers. We will use this list as the `for` statement's list. In the shell, type `for i in range(10):` and press Enter. Nothing will happen, but the shell will indent the cursor, because it is waiting for you to type in the for-block. Type `print i` and press Enter. Then, to tell the interactive shell you are done typing in the for-block, press Enter again to enter a blank line. The shell will then execute your `for` statement and block:

```
>>> for i in range(10):
        print i


0
1
2
3
4
5
6
7
8
9
>>>
```

The `for` loop executes the code inside the for-block once for each item in the list. Each time it executes the code in the for-block, the variable `i` is assigned the next value of the next item in the list. If we used the `for` statement with the list `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` instead of `range(10)`, it would have been the same since the `range()` function's return value is the same as that list:

```
>>> for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
	print i


0
1
2
3
4
5
6
7
8
9
>>>
```

Try typing this into the shell: `for thing in ['cats', 'pasta', 'programming', 'spam']:` and press Enter, then type `print 'I really like ' + thing` and press Enter, and then press Enter again to tell the shell to end the for-block. The output should look like this:

```
>>> for thing in ['cats', 'pasta', 'programming', 'spam']:
	print 'I really like ' + thing


I really like cats
I really like pasta
I really like programming
I really like spam
>>>
```

And remember, because strings are also a sequence data type just like lists, you can use them in `for` statements as well. This example uses a single character from the string on each iteration:

```
>>> for i in 'Hello world!':
	print i


H
e
l
l
o
 
w
o
r
l
d
!
>>>
```

The `for` loop is very similar to the `while` loop, but when you only need to iterate over items in a list, using a `for` loop is much less code to type. You can make a `while` loop that acts the same way as a `for` loop by adding extra code:

```
>>> sequence = ['cats', 'pasta', 'programming', 'spam']
>>> index = 0
>>> while (index < len(sequence)):
	thing = sequence[index]
	print 'I really like ' + thing
	index = index + 1


I really like cats
I really like pasta
I really like programming
I really like spam
>>> 
```

But using the `for` statement automatically does all this extra code for us and makes programming easier since we have less to type. Our Hangman game will use `for` loops so you can see how useful they are in real games.

One more thing about `for` loops, is that the `for` statement has the `in` keyword in it. But when you use the `in` keyword in a `for` statement, Python does not treat it like the `in` operator you would use in something like `42 in [0, 42, 67]`. The `in` keyword in `for` statements is just used to separate the variable and the list it gets its values from.

Code Explanation continued...

```
    print 'Missed letters:',
    for letter in missedLetters:
        print letter,
    print
```

This `for` loop will display all the missed guesses that the player has made. When you play Hangman on paper, you usually write down these letters off to the side so you know not to guess them again. On each iteration of the loop the value of `letter` will be each letter in `missedLetters` in turn. Remember that a comma at the end of the `print` statement will make it print a space instead of a "newline" character, so all the missed letters will be on the same line.

If `missedLetters` was `'ajtw'` then this `for` loop would display `a j t w`.

So by this point we have shown the player the hangman board and the missed letters. Now we want to

print the secret word, except we want blank lines for the letters. We can use the _ character (called the underscore character) for this. But we should print the letters in the secret word that the player has guessed, and use _ characters for the letters the player has not guessed yet. We can first create a string with nothing but one underscore for each letter in the secret word. Then we can replace the blanks for each letter in `correctLetters`. So if the secret word was `'otter'` then the blanked out string would be `'_____'` (five _ characters). If `correctLetters` was the string `'rt'` then we would want to change the blanked string to `'_tt_r'`. Here is the code that does that:

```
114.     blanks = '_' * len(word)
115.
116.     for i in range(len(secretWord)): # replace blanks
    with correctly guessed letters
117.         if word[i] in correctLetters:
118.             blanks = blanks[:i] + word[i] + blanks
    [i+1:]
```

Line 114 creates the `blanks` variable full of _ underscores using string replication. Remember that the `*` operator can also be used on a string and an integer, so the expression `'hello' * 3` evaluates to `'hellohellohello'`. This will make sure that `blanks` has the same number of underscores as `secretWord` has letters.

Then we use a `for` loop to go through each letter in `secretWord` and replace the underscore with the actual letter if it exists in `correctLetters`. Line 118 may look confusing. It seems that we are using the square brackets with the `blanks` and `secretWord` variables. But wait a second, `blanks` and `secretWord` are strings, not lists. And the `len()` function also only takes lists as parameters, not strings. But in Python, many of the things you can do to lists you can also do to strings:

Strings Act Like Lists

Surprise! Strings act a lot like lists. In fact, almost all of the things you can do on lists you can also do on strings. Just think of strings as "lists" of one-letter strings. So `'Hello'` acts similar to `['H', 'e', 'l', 'l', 'o']`. (They are still different values and have different data types though. `'Hello' == ['H', 'e', 'l', 'l', 'o']` would be `False`.)) The square brackets can also pick out individual characters from a string just like it can pick out individual items from a list. In the interactive shell, type `fizz = 'Hello world!'` and then `fizz[0]`:

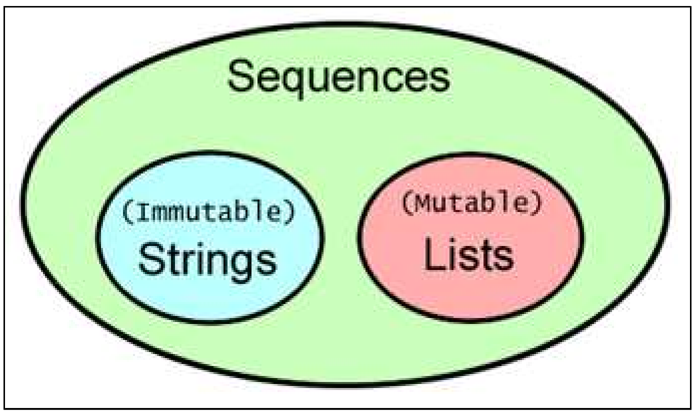
```
>>> fizz = 'Hello world!'
>>> fizz[0]
'H'
>>>
```

You can also find out how many characters are in a string with the `len()` function. Type in `len`

(fizz) into the shell:

```
>>> fizz = 'Hello world!'
>>> fizz[0]
'H'
>>> len(fizz)
12
>>>
```

However, you cannot change a character in a string or remove a character with del statement. This is because a list is a **mutable sequence** and a string is an **immutable sequence**. "Mutable" is another word for "changeable." The word "immutable" means "cannot be changed." A sequence is a series of things (like in real life, a dance sequence is a series of different dance steps done one after another). The reason strings are immutable and lists are mutable has to do with how the Python interpreter is programmed, but it isn't important for us to know in order to make games. If we want to change a string, we can create a copy of the string with slices (explained next) the same way we do on line 118.



So remember, you can use index assignment or del with lists but not with strings.

List Slicing and Substrings

Slicing is like indexing with multiple indexes instead of just one. Instead of putting one index in between the square brackets, we put two indexes separated by a colon. To grab the first three items from our animals list with animals[0:3], which means "all items in animals from item 0 up to (but not including) item 3."

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[0:3]
['aardvark', 'anteater', 'antelope']
>>>
```

To grab items 2 and 3 from animals, use the slice animals[2:4]. Try typing it into the shell:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[2:4]
['antelope', 'albert']
>>>
```

You can use slicing to get a part of a string (called a **substring** from a string. Try typing `'Hello world!'[3:8]` into the shell:

```
>>> 'Hello world!'[3:8]
'lo wo'
>>>
```

So remember that on the right side of any list or string value (or a variable that contains a list or string value), you can put square brackets to extract a single or several items from the **sequence**. ("Sequence" refers to a group of data types that include strings and lists.)

Code Explanation continued...

```
    for i in range(len(secretWord)): # replace blanks with correctly guessed letters
        if secretWord[i] in correctLetters:
            blanks = blanks[:i] + secretWord[i] + blanks[i+1:]
```

Let's pretend the value of `secretWord` is `'otter'` and the value in `correctLetters` is `'tr'`. Then `len(secretWord)` will return `5`. Then `range(len(secretWord))` becomes `range(5)`, which in turn returns the list `[0, 1, 2, 3, 4]`.

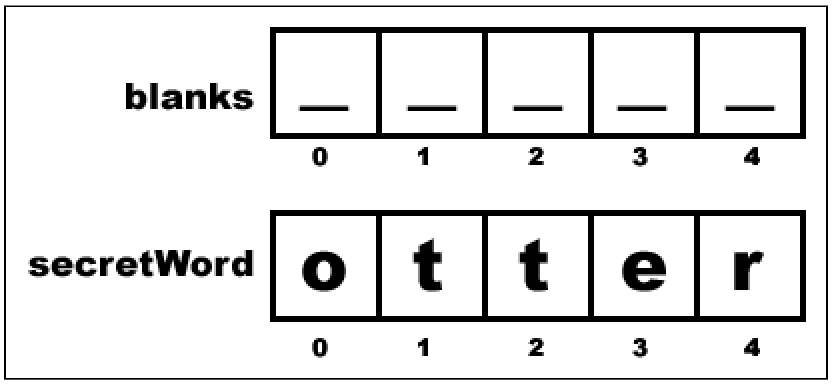
Because the value of `i` will take on each value in `[0, 1, 2, 3, 4]`, then the `for` loop code is equivalent to this:

```
    if secretWord[0] in correctLetters:
        blanks = blanks[:0] + secretWord[0] + blanks[1:]
    if secretWord[1] in correctLetters:
        blanks = blanks[:1] + secretWord[1] + blanks[2:]
    if secretWord[2] in correctLetters:
        blanks = blanks[:2] + secretWord[2] + blanks[3:]
    if secretWord[3] in correctLetters:
        blanks = blanks[:3] + secretWord[3] + blanks[4:]
    if secretWord[4] in correctLetters:
```

```
        blanks = blanks[:4] + secretWord[4] + blanks[5:]
```

(By the way, writing out the code like this instead of using a loop is called **loop unrolling**.)

If you are confused as to what the value of something like `secretWord[0]` or `blanks[3:]` is, then look at this picture. It shows the value of the `secretWord` and `blanks` variables, and the index for each letter in the string.



If we replace the list slices and the list indexes with the values that they represent, the unrolled loop code would be the same as this:

```
    if 'o' in 'tr':                # Condition is False,
blanks == '_____'
        blanks = '' + 'o' + '____' # This line is skipped.
    if 't' in 'tr':                # Condition is True,
blanks == '_____'
        blanks = '_' + 't' + '___' # This line is executed.
    if 't' in 'tr':                # Condition is True,
blanks == '_t___'
        blanks = '_t' + 't' + '__' # This line is executed.
    if 'e' in 'tr':                # Condition is False,
blanks == '_tt__'
        blanks = '_tt' + 'e' + '_' # This line is skipped.
    if 'r' in 'tr':                # Condition is True,
blanks == '_tt__'
        blanks = '_tt_' + 'r' + '' # This line is executed.

    # blanks now has the value '_tt_r'
```

The above three boxes of code all do the *same thing* (at least, they do when `secretWord` is `'otter'` and `correctLetters` is `'tr'`. The first box is the actual code we have in our game. The second box shows code that does the same thing except without a `for` loop. The third box is the same as the second box, except we have evaluated many of the expressions in the second box.

The next few lines of code display the new value of `blanks` with spaces in between each letter.

```
    for letter in blanks: # show the secret word with spaces in between each letter
        print letter,
    print
```

This `for` loop will print out each character in the string `blanks`. Remember that by now, `blanks` may have some of its underscores replaced with the letters in `secretWord`. The comma at the end of the `print` statement causes it to display a space instead of a newline character.

This is the end of the `displayBoard()` function.

```
def getGuess(alreadyGuessed):
    # Returns the letter the player entered. This function makes sure the player entered a single letter, and not something else.
```

The `getGuess()` function has a string parameter called `alreadyGuessed` which contains the letters the player has already guessed, and will ask the player to guess a single letter. This single letter will be the return value for this function.

```
    while True:
        print 'Guess a letter.'
        guess = raw_input()
        guess = guess.lower()
```

We will use a `while` loop because we want to keep asking the player for a letter until they enter text that is a single letter they have not guessed previously. Notice that the condition for the `while` loop is simply the boolean value `True`. That means the only way execution will ever leave this loop is by executing a `break` statement (which leaves the loop) or a `return` statement (which leaves the entire function).

The code inside the loop asks the player to enter a letter, which is stored in the variable `guess`. If the player entered a capitalized letter, it will be converted to lowercase on line 129.

elif ("Else If") Statements

Take a look at the following code:

```
if catName == 'Fuzzball':
    print 'Your cat is fuzzy.'
else:
    print 'Your cat is not very fuzzy at all.'
```

We've seen code like this before and it's rather simple. If the `catName` variable is equal to the string `'Fuzzball'`, then the `if` statement's condition is `True` and we tell the user that her cat is fuzzy. If `catName` is anything else, then we tell the user her cat is not fuzzy.

But what if we wanted something else besides "fuzzy" and "not fuzzy"? We could put another `if` and `else` statement inside the first `else` block like this:

```
if catName == 'Fuzzball':
    print 'Your cat is fuzzy.'
else:
    if catName == 'Spots':
        print 'Your cat is spotted.'
    else:
        print 'Your cat is neither fuzzy nor spotted.'
```

But if we wanted more things, then the code starts to have a lot of indentation:

```
if catName == 'Fuzzball':
    print 'Your cat is fuzzy.'
else:
    if catName == 'Spots'
        print 'Your cat is spotted.'
    else:
        if catName == 'FattyKitty'
            print 'Your cat is fat.'
        else:
            if catName == 'Puff'
                print 'Your cat is puffy.'
            else:
                print 'Your cat is neither fuzzy nor spotted
nor fat nor puffy.'
```

Typing all those spaces means you have more chances of making a mistake with the indentation. So Python has the `elif` keyword. Using `elif`, the above code looks like this:

```
if catName == 'Fuzzball':
    print 'Your cat is fuzzy.'
elif catName == 'Spots'
    print 'Your cat is spotted.'
elif catName == 'FattyKitty'
    print 'Your cat is fat.'
elif catName == 'Puff'
    print 'Your cat is puffy.'
else:
    print 'Your cat is neither fuzzy nor spotted nor fat nor puffy.'
```

If the condition for the `if` statement is `False`, then the program will check the condition for the first `elif` statement (which is `catName == 'Spots'`. If that condition is `False`, then the program will check the condition of the next `elif` statement. If ALL of the conditions for the `if` and `elif` statements are `False`, then the code in the `else` block executes.

But if one of the `elif` conditions are `True`, the elif-block code is executed and then execution jumps down to the first line past the else-block. So *only one* of the blocks in this if-elif-else statement will be executed. You can also leave off the else-block if you don't need one, and just have an if-elif statement.

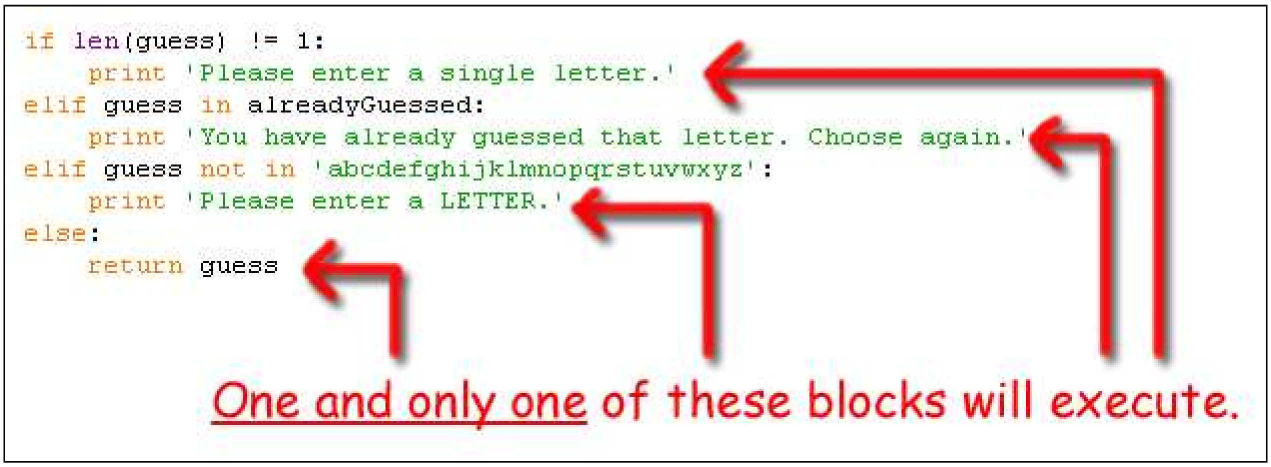
Code Explanation continued...

```
130.         if len(guess) != 1:
131.             print 'Please enter a single letter.'
132.         elif guess in alreadyGuessed:
133.             print 'You have already guessed that letter. Choose again.'
134.         elif guess not in 'abcdefghijklmnopqrstuvwxyz':
135.             print 'Please enter a LETTER.'
136.         else:
137.             return guess
```

The `guess` variable contains the text the player typed in for their guess. We need to make sure they typed in a single lowercase letter. If they didn't, we should loop back and ask them again. The `if` statement's condition checks that the text is one and only letter. If it is not, then we execute the if-block code, and then execution jumps down past the else-block. But since there is no more code after this if-elif-else statement, execution loops back to line 126.

If the condition for the `if` statement is `False`, we check the `elif` statement's condition on line 132. This condition is `True` if the letter exists inside the `alreadyGuessed` variable (remember, this is a string that has every letter the player has already guessed). If this condition is `True`, then we display the error message to the player, and jump down past the else-block. But then we would be at the end of the while-block, so execution jumps back up to line 126.

If the condition for the `if` statement and the `elif` statement are both `False`, then we check the second `elif` statement's condition on line 134. If the player typed in a number or a funny character (making `guess` have a value like `'5'` or `'!'`), then `guess` would not exist in the string `'abcdefghijklmnopqrstuvwxyz'`. If this is the case, the `elif` statement's condition is `True`



Unless these three conditions are all `False`, the player will keep looping and keep being asked for a letter. But when all three of the conditions are `False`, then the else-block's `return` statement will run and we will exit this loop and function.

```
139. def playAgain():
140.     # This function returns True if the player wants
     to play again, otherwise it returns False.
141.     print 'Do you want to play again? (yes or no)'
142.     return raw_input().lower().startswith('y')
```

The `playAgain()` function has just a `print` statement and a `return` statement. The `return` statement has an expression that looks complicated, but we can break it down. Once we evaluate this expression to a value, that value will be returned from this function.

The expression on line 142 doesn't have any operators, but it does have a function call and two method calls. The function call is `raw_input()` and the method calls are `lower()` and `startswith('y')`. Remember that method calls are function calls that are attached by a period to the *value* on their *left*. `lower()` is attached to the return value of `raw_input()`.

`raw_input()` returns a string of the text that the user typed in. Here's a step by step look at how Python evaluates this expression if the user types in YES.

```
return raw_input().lower().startswith('y')
        ↓
return 'YES'.lower().startswith('y')
        ↓
return 'yes'.startswith('y')
        ↓
return True
```

The point of the `playAgain()` function is to let the player type in yes or no to tell our program if they want to play another round of Hangman. If the player types in YES, then the return value of `raw_input()` is the string `'YES'`. `'YES'.lower()` returns the lowercase version of the attached string. So the return value of `'YES'.lower()` is `'yes'`.

But there's the second method call, `startswith('y')`. This function returns `True` if the associated string begins with the string parameter between the parentheses, and `False` if it doesn't. The return value of `'yes'.startswith('y')` is `True`.

Now we have evaluated this expression! We can see that what this does is let the player type in a response, we lowercase the response, check if it begins with the letter `'y'` or `'Y'`, and then return `True` if it does and `False` if it doesn't. Whew!

On a side note, there is also a `endswith(someString)` string method that will return `True` if the string ends with the string in `someString` and `False` if it doesn't.

Code Explanation continued...

That's all the functions we are creating for this game! `getRandomWord(wordList)` will take a list of strings passed to it as a parameter, and return one string from it. That is how we will choose a word for the player to guess.

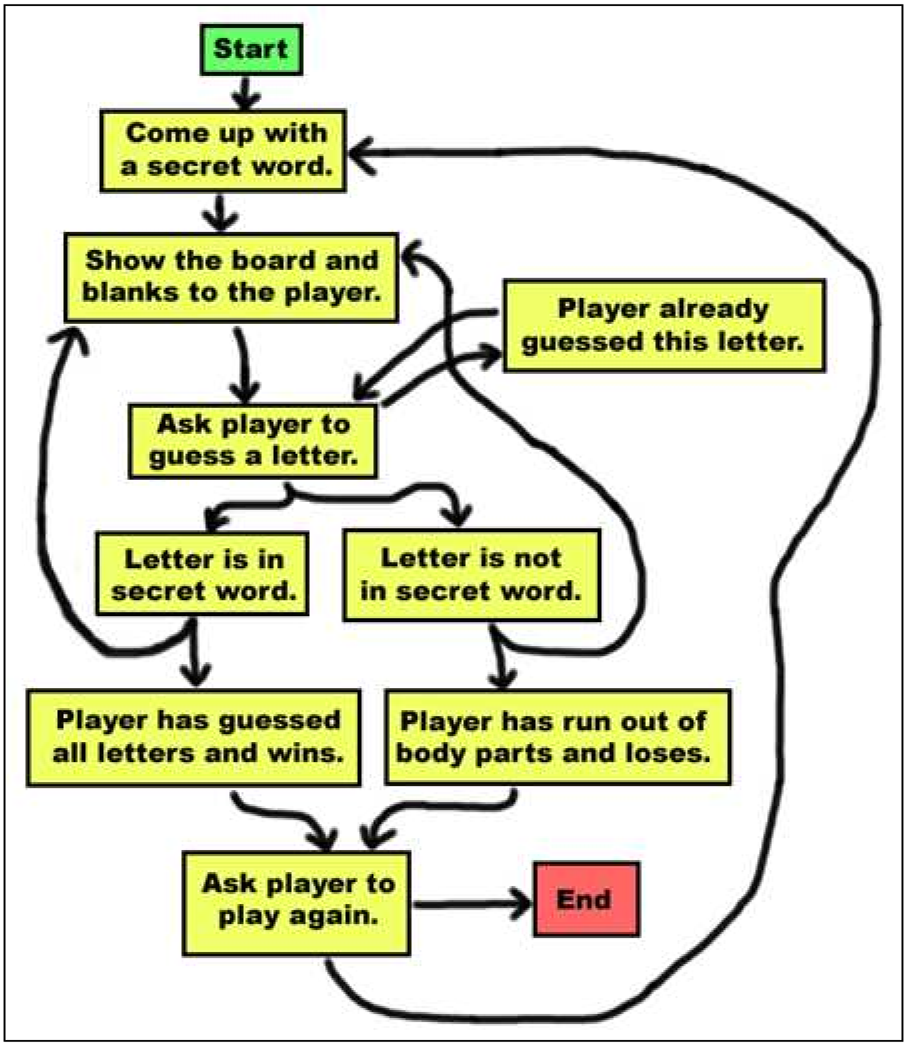
`displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)` will show the current state of the board, including how much of the secret word the player has guessed so far and the wrong letters the player has guessed. This function needs four parameters passed to work correctly. `HANGMANPICS` is a list of strings that hold the ASCII art for each possible hangman board. `correctLetters` and `missedLetters` are strings made up of the letters that the player has guessed that are in and not in the secret word. And `secretWord` is the secret word the player is trying to guess. This function has no return value.

`getGuess(alreadyGuessed)` takes a string of letters the player has already guessed and will

keep asking the player for a letter that is a letter that he hasn't already guessed. (That is, a letter that is not in `alreadyGuessed`. This function returns the string of the acceptable letter the player guessed.

`playAgain()` is a function that asks if the player wants to play another round of Hangman. This function returns `True` if the player does and `False` if the player doesn't.

We'll now start the code for the main part of the game, which will call the above functions as needed. Look back at our flow chart.



We need to write code that does everything in this flow chart, and does it in order. The main part of the code starts at line 145:

```
145. print 'H A N G M A N'
146. missedLetters = ''
147. correctLetters = ''
```

```
148. secretWord = getRandomWord(words)
149. gameIsDone = False
```

Line 145 is the first actual line that executes in our game. Everything previous was just function definitions and a very large variable assignment for `HANGMANPICS`. We start by assigning a blank string for `missedLetters` and `correctLetters`, because the player has not guessed any missed or correct letters yet. Then we call `getRandomWord(words)`, where `words` is a variable with the huge list of possible secret words we assigned on line 98. The return value of `getRandomWord(words)` is one of these words, and we save it to the `secretWord` variable. Then we also set a variable named `gameIsDone` to `False`. We will set `gameIsDone` to `True` when we want to signal that the game is over and the program should ask the player if they want to play again.

Setting the values of these variables is what we do before the player starts guessing letters.

```
151. while True:
152.     displayBoard(HANGMANPICS, missedLetters,
     correctLetters, secretWord)
```

The `while` loop's condition is always `True`, which means we will always loop forever until a `break` statement is encountered. We will execute a `break` statement when the game is over (either because the player won or the player lost).

Line 152 calls our `displayBoard()` function, passing it the list of hangman ASCII art pictures and the three variables we set on lines 146, 147, and 148. Program execution moves to the start of `displayBoard()` at line 105. Based on how many letters the player has correctly guessed and missed, this function displays the appropriate hangman board to the player.

```
154.     # Let the player type in a letter.
155.     guess = getGuess(missedLetters + correctLetters)
```

If you look at our flow chart, you see only one arrow going from the "Show the board and the blanks to the player." box to the "Ask a player to guess a letter." box. Since we have already written a function to get the guess from the player, let's call that function. Remember that the function needs all the letters in `missedLetters` and `correctLetters` combined, so we will pass as an argument a string that is a concatenation of both of those strings. This argument is needed by `getGuess()` because the function

has code to check if the player types in a letter that they have already guessed.

```
    if guess in secretWord:
        correctLetters = correctLetters + guess
```

Now let's see if the single letter in the `guess` string exists in `secretWord`. If it does exist, then we should concatenate the letter in `guess` to the `correctLetters` string. Next we can check if we have guessed all of the letters and won.

```
        # Check if the player has won
        foundAllLetters = True
        for i in range(len(secretWord)):
            if secretWord[i] not in correctLetters:
                foundAllLetters = False
                break
```

How do we know if the player has guessed every single letter in the secret word? Well, `correctLetters` has each letter that the player correctly guessed and `secretWord` is the secret word itself. We can't just check if `correctLetters == secretWord` because consider this situation: if `secretWord` was the string `'otter'` and `correctLetters` was the string `'orte'`, then `correctLetters == secretWord` would be `False` even though the player has guessed each letter in the secret word.

The player simply guessed the letters out of order and they still win, but our program would incorrectly think the player hasn't won yet. Even if they did guess the letters in order, `correctLetters` would be the string `'oter'` because the player can't guess the letter t more than once. The expression `'otter' == 'oter'` would evaluate to `False` even though the player won.

The only way we can be sure the player won is to go through each letter in `secretWord` and see if it exists in `correctLetters`. If, and only if, every single letter in `secretWord` exists in `correctLetters` will the player have won.

Note that this is different than checking if every letter in `correctLetters` is in `secretWord`. If `correctLetters` was the string `'ot'` and `secretWord` was `'otter'`, it would be true that every letter in `'ot'` is in `'otter'`, but that doesn't mean the player has guessed the secret word and won.

So how can we do this? We can loop through each letter in `secretWord` and if we find a letter that does not exist in `correctLetters`, we know that the player has not guessed all the letters. This is why we create a new variable named `foundAllLetters` and set it to the boolean value `True`. We start out assuming that we have found all the letters, but will change `foundAllLetters` to `False` when we find a letter in `secretWord` that is not in `correctLetters`.

The for loop will go through the numbers 0 up to (but not including) the length of the word. Remember that `range(5)` will evaluate to the list `[0, 1, 2, 3, 4, 5]`. So on line 162, the program executes all the code inside the for-block with the variable `i` will be set to `0`, then `1`, then `2`, then `3`, then `4`, then `5`.

We use `range(len(secretWord))` so that `i` can be used to access each letter in the secret word. So if the first letter in `secretWord` (which is located at `secretWord[0]`) is not in `correctLetters`, we know we can set `foundAllLetters` to `False`. Also, because we don't have to check the rest of the letters in `secretWord`, we can just break out of this loop. Otherwise, we loop back to line 163 and check the next letter.

If `foundAllLetters` manages to survive every single letter without being turned to `False`, then it will keep the original `True` value we gave it. Either way, the value in `foundAllLetters` is accurate by the time we get past this for loop and run line 166.

```
166.         if foundAllLetters:
167.             print 'Yes! The secret word is "' +
    secretWord + '"! You have won!'
168.             gameIsDone = True
```

This is a simple check to see if we found all the letters. If we have found every letter in the secret word, we should tell the player that they have won. We will also set the `gameIsDone` variable to `True`. We will check this variable to see if we should let the player guess again or if the player is done guessing.

```
169.     else:
```

This is the start of the else-block. Remember, the code in this block will execute if the condition was `False`. But which condition? To find out, point your finger at the start of the `else` keyword and move it straight up. You will see that the `else` keyword's indentation is the same as the `if` keyword's indentation on line 157. So if the condition on line 157 was `False`, then we will run the code in this

else-block. Otherwise, we skip down past the else-block to line 177.

```
170.         missedLetters = missedLetters + guess
```

Because the player's guessed letter was wrong, we will add it to the `missedLetters` string. This is like what we did on line 158 when the player guessed correctly.

```
172.         # Check if player has guessed too many times and lost
173.         if len(missedLetters) == len(HANGMANPICS) - 1:
174.             displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)
175.             print 'You have run out of guesses!\nAfter ' + str(len(missedLetters)) + ' missed guesses and ' + str(len(correctLetters)) + ' correct guesses, the word was "' + secretWord + '"'
176.             gameIsDone = True
```

Think about how we know when the player has guessed too many times. When you play Hangman on paper, this is when the drawing of the hangman is finished. We draw the hangman on the screen with print statements, based on how many letters are in `missedLetters`. Remember that each time the player guesses wrong, we add (or as a programmer would say, concatenate) the wrong letter to the string in `missedLetters`. So the length of `missedLetters` (or, in code, `len(missedLetters)`) can tell us the number of wrong guesses.

At what point does the player run out of guesses and lose? Well, the `HANGMANPICS` list has 7 pictures (really, they are ASCII art strings). So when `len(missedLetters)` equals `6`, we know the player has lost because the hangman picture will be finished. (Remember that `HANGMANPICS[0]` is the first item in the list, and `HANGMANPICS[6]` is the last one. This is because the index of a list with 7 items goes from 0 to 6, not 1 to 7.)

So why do we have `len(missedLetters) == len(HANGMANPICS) - 1` as the condition on line 173, instead of `len(missedLetters) == 6`? Pretend that we add another string to the `HANGMANPICS` list (maybe a picture of the full hangman with a tail, or a third mutant arm). Then the last picture in the list would be at `HANGMANPICS[7]`. So not only would we have to change the `HANGMANPICS` list with a new string, but we would also have to remember to change line 173 to `len(missedLetters) == 7`. This might not be a big deal for a

program like Hangman, but when you start writing larger programs you may have to change several different lines of code all over your program just to make a
change in the program's behavior. This way, if we want to make the game harder or easier, we just have to add or remove ASCII art strings to `HANGMANPICS` and change nothing else.

A second reason we user `len(HANGMANPICS) - 1` is so that when we read the code in this program later, we know why this program behaves the way it does. If you wrote `len(missedLetters) == 6` and then looked at the code two weeks later, you may wonder what is so special about the number 6. You may have forgotten that 6 is the last index in the HANGMANPICS list. Of course, you could write a comment to remind yourself, like:

```
        if len(missedLetters) == 6: # 6 is the last index in the HANGMANPICS list
```

But it is easier to just use `len(HANGMANPICS) - 1` instead.

So, when the length of the `missedLetters` string is equal to `len(HANGMANPICS) - 1`, we know the player has run out of guesses and has lost the game. We print a long string telling the user what the secret word was, and then set the `gameIsDone` value to the boolean value `True`. This is how we will tell ourselves that the game is done and we should start over.

Remember that when we have \n in a string, that represents the newline character.

```
    # Ask the player if they want to play again (but only if the game is done).
    if gameIsDone:
        if playAgain():
            missedLetters = ''
            correctLetters = ''
            gameIsDone = False
            secretWord = getRandomWord(words)
```

If the player won or lost after guessing their letter, then our code would have set the `gameIsDone` variable to `True`. If this is the case, we should ask the player if they want to play again. We already wrote the `playAgain()` function to handle getting a yes or no from the player. This function returns a boolean value of `True` if the player wants to play another game of Hangman, and `False` if they've had enough.

If the player does want to play again, we will reset the values in `missedLetters` and `correctLetters` to blank strings, set `gameIsDone` to `False`, and then choose a new secret word by calling `getRandomWord()` again, passing it the list of possible secret words.

This way, when we loop back to the beginning of the loop (on line 151) the board will be back to the start (remember we decide which hangman picture to show based on the length of `missedLetters`, which we just set as the blank string) and the game will be just as the first time we entered the loop. The only difference is we will have a new secret word, because we programmed `getRandomWord()` to return a randomly chosen word each time we call it.

There is a small chance that the new secret word will be the same as the old secret word, but this is just a coincidence. Let's say you flipped a coin and it came up heads, and then you flipped the coin again and it also came up heads. Both coin flips were random, it was just a coincidence that they came up the same both times. Accordingly, you may get the same word return from `getRandomWord()` twice in a row, but this is just a coincidence.

```
        else:
            break
```

If the player typed in `'no'` when asked if they wanted to play again, then they return value of the call to the `playAgain()` function would be `False` and the else-block would have executed. This else-block only has one line, a `break` statement. This causes the execution to jump to the end of the loop that was started on line 151. But because there is no more code after the loop, the program terminates.

And that's it!

This program was much bigger than the Dragon World program, but this program is also more sophisticated. It really helps to make a flow chart or small sketch to remember how you want everything to work. Take a look at the flow chart and try to find the lines of code that represent each block.

At this point, you can move on to the next chapter. But I suggest you keep reading on to find out about some ways we can improve our Hangman game.

After you have played Hangman a few times, you might think that six guesses aren't enough to get many of the words. We can easily give the player more guesses by adding more multi-line strings to the `HANGMANPICS` list. It's easy, just change the] square bracket on line 96 to a ,''' comma and three quotes (see line 96 below). Then add the following:

```
96. ==============''', '''
97.
```

```
 98.
 99.     +------+
100.     |      |
101.     |      |
102.    [O      |
103.    /|\     |
104.   / | \    |
105.     |      |
106.    / \     |
107.   /   \    |
108.            |
109.            |
110. =============='''
111.
112.
113.     +------+
114.     |      |
115.     |      |
116.    [O]     |
117.    /|\     |
118.   / | \    |
119.     |      |
120.    / \     |
121.   /   \    |
122.            |
123.            |
124. ==============''']
```

We have added two new multi-line strings to the `HANGMANPICS` list, one with the hangman's left ear drawn, and the other with both ears drawn. Because our program will tell the player they have lost when the number of guesses is the same as the number of strings in `HANGMANPICS` (minus one), this is the only change we need to make.

We can also change the list of words by changing the words on line 98. Instead of animals, we could have colors, shapes, or fruits:

```
98. words = 'red orange yellow green blue indigo violet
    white black brown'.split()
```

```
98. words = 'square triangle rectangle circle ellipse
```

```
rhombus trapazoid chevron pentagon hexagon septagon
octogon'.split()
```

```
words = 'apple orange lemon lime pear watermelon grape
grapefruit cherry banana cantalope mango strawberry
tomato'.split()
```

Dictionaries

With some modification, we can change our code so that our Hangman game can use all of these words as separate sets. We can tell the player which set the secret word is from (like "animal", "color", "shape", or "fruit"). This way, the player isn't guessing animals all the time.

To make this change, we will introduce a new data type called a dictionary. A **dictionary** is a collection of other values much like a list, but instead of accessing the items in the dictionary with an integer index, you access them with an index of any data type (but most often strings).

Try typing the following into the shell:

```
stuff = {'hello':'Hello there, how are you?', 'chat':'How is the weather?', 'goodbye':'It was nice talking to you!'}
```

Those are curly braces { and }. On the keyboard they are on the same key as the square braces [and]. We use curly braces to type out a dictionary value in Python. The values in between them are **key-value pairs**. The keys are the things on the left of the colon and the values are on the right of the colon. You can access the values (which are like items in lists) in the dictionary by using the key (which are like indexes in lists). Try typing into the shell `stuff['hello']` and `stuff['chat']` and `stuff['goodbye']`:

```
>>> stuff = {'hello':'Hello there, how are you?', 'chat':'How is the weather?',
'goodbye':'It was nice talking to you!'}
>>> stuff['hello']
'Hello there, how are you?'
>>> stuff['chat']
'How is the weather?'
>>> stuff['goodbye']
'It was nice talking to you!'
>>>
```

You see, instead of putting an integer index in between the square brackets, you put a key string index. This will evaluate to the value for that key. You can get the size (that is, how many key-value pairs in the

dictionary) with the `len()` function. Try typing `len(stuff)` into the shell:

```
>>> len(stuff)
3
>>>
```

The list version of this dictionary would have only the values, and look something like this:

```
listStuff = ['Hello there, how are you?', 'How is the
weather?', 'It was nice talking to you!']
```

The list doesn't have any keys, like `'hello'` and `'chat'` and `'goodbye'` in the dictionary. We have to use integer indexes `0`, `1`, and `2`.

Dictionaries are different from lists because they are **unordered**. The first item in a list named `listStuff` would be `listStuff[0]`. But there is no "first" item in a dictionary, because dictionaries do not have any sort of order. Try typing this into the shell:

```
favorites1 = {'fruit':'apples', 'animal':'cats',
'number':42}
favorites2 = {'animal':'cats', 'number':42,
'fruit':'apples'}
favorites1 == favorites2
```

```
>>> favorites1 = {'fruit':'apples', 'animal':'cats', 'number':42}
>>> favorites2 = {'animal':'cats', 'number':42, 'fruit':'apples'}
>>> favorites1 == favorites2
True
>>>
```

As you can see, the expression `favorites1 == favorites2` evaluates to `True` because dictionaries are unordered, and they are considered to be the same if they have the same key-value pairs in them. Lists are ordered, so a list with the same values in them but in a different order are not the same. Try typing this into the shell:

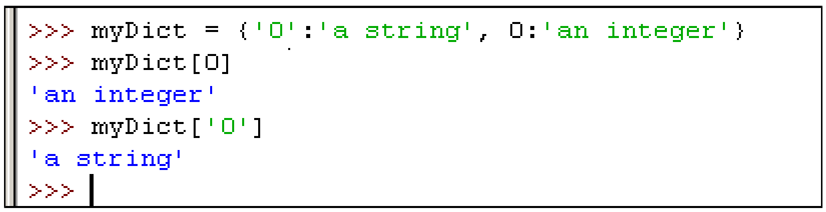
```
listFavs1 = ['apples', 'cats', 42]
listFavs2 = ['cats', 42, 'apples']
listFavs1 == listFavs2
```

```
>>> listFavs1 = ['apples', 'cats', 42]
>>> listFavs2 = ['cats', 42, 'apples']
>>> listFavs1 == listFavs2
False
>>>
```

As you can see, the two lists `listFavs1` and `listFavs2` are not considered to be the same because order matters in lists.

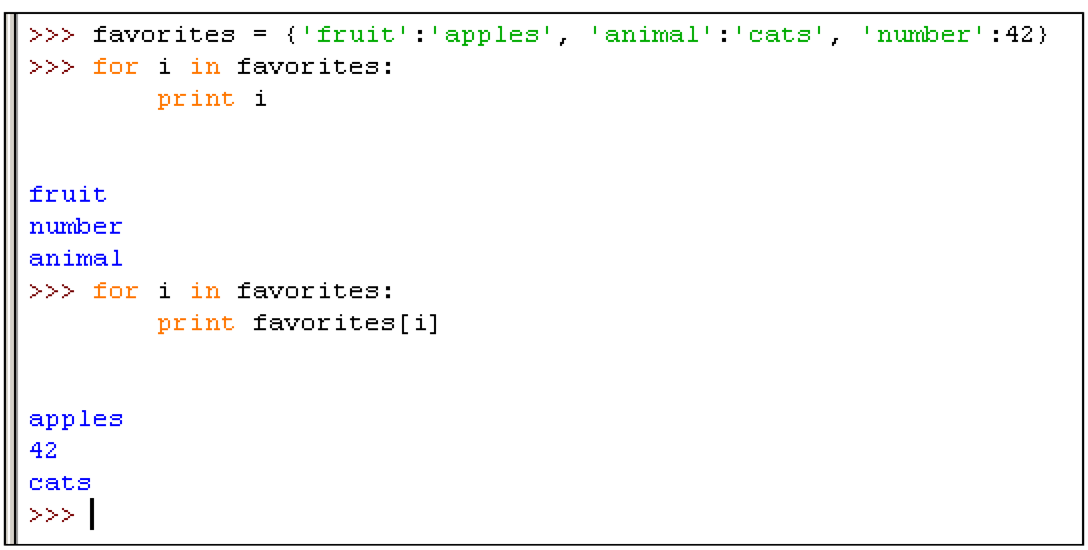
You can also use integers as the keys for dictionaries. Dictionaries can have keys of any data type, not just strings. But remember, because `0` and `'0'` are different values, they will be different keys. Try typing this into the shell:

```
myDict = {'0':'a string', 0:'an integer'}
myDict[0]
myDict['0']
```



You might think that using a `for` loop is hard with dictionaries because they do not have integer indexes. But actually, it's easy. Try typing the following into the shell. (Here's a hint, in IDLE, you do not have to type spaces to start a new block. IDLE does it for you. To end the block, just insert a blank line by just hitting the Enter key. Or you could start a new file, type in this code, and then press F5 to run the program.)

```
favorites = {'fruit':'apples', 'animal':'cats', 'number':42}
for i in favorites:
    print i
for i in favorites:
    print favorites[i]
```



As you can see, if you just use a dictionary in a `for` loop, the variable `i` will take on the values of the dictionary's *keys*, not its values. But if you have the dictionary and the key, you can get the value as we do above with `favorites[i]`. But remember that because dictionaries are unordered, you cannot predict which order the `for` loop will execute in. Above, we typed the `'animal'` key as coming before the `'number'` key, but the `for` loop printed out `'number'` before `'animal'`.

Dictionaries also have two useful methods, `keys()` and `values()`. These will return (ordered) lists of the key values and the value values, respectively. Try typing the following into the shell:

```
favorites = {'fruit':'apples', 'animal':'cats', 'number':42}
favorites.keys() favorites.values()
```

```
>>> favorites = {'fruit':'apples', 'animal':'cats', 'number':42}
>>> favorites.keys()
['fruit', 'number', 'animal']
>>> favorites.values()
['apples', 42, 'cats']
>>>
```

Using these methods to get a list of the keys and values that are in a dictionary can be very helpful.

Sets of Words for Hangman

So how can we use dictionaries in our game? First, let's change the list `words` into a dictionary whose keys are strings and values are lists of strings. (Remember that the string method `split()` evaluates to a list.

```
words = {'Colors':'red orange yellow green blue indigo
    violet white black brown'.split(),
'Shapes':'square triangle rectangle circle ellipse
    rhombus trapazoid chevron pentagon hexagon septagon
    octogon'.split(),
'Fruits':'apple orange lemon lime pear watermelon
    grape grapefruit cherry banana cantalope mango
    strawberry tomato'.split(),
'Animals':'bat bear beaver cat cougar crab deer dog
    donkey duck eagle fish frog goat leech lion lizard
    monkey moose mouse otter owl panda python rabbit rat
    shark sheep skunk squid tiger turkey turtle weasel
    whale wolf wombat zebra'.split()}
```

This code is put across multiple lines in the file, even though the Python interpreter thinks of it as just one "line of code." (The line of code doesn't end until the final } curly brace.)

Now we will have to change our `getRandomWord()` function so that it chooses a random word from a dictionary of lists of strings, instead of from a list of strings. Here is what the function originally looked like:

```
100. def getRandomWord(wordList):
101.     # This function returns a random string from the passed list of strings.
102.     wordIndex = random.randint(0, len(wordList) - 1)
103.     return wordList[wordIndex]
```

Change the code in this function so that it looks like this:

```
100. def getRandomWord(wordDict):
101.     # This function returns a random string from the passed dictionary of lists of strings, and the key also.
102.
103.     # First, randomly select a key from the dictionary:
104.     wordKey = random.choice(wordDict.keys())
105.
106.     # Second, randomly select a word from the key's list in the dictionary:
107.     wordIndex = random.randint(0, len(wordDict[wordKey]) - 1)
108.
109.     return [wordDict[wordKey][wordIndex], wordKey]
```

Line 100 just changes the name of the parameter to something a little more descriptive. Now instead of choosing a random word from a list of strings, first we choose a random key from the dictionary and then we choose a random word from the key's list of strings. Line 104 calls a new function in the `random` module named `choice()`. The `choice()` function has one parameter, a list. The return value of `choice()` is an item randomly selected from this list each time it is called.

Remember that `randint(a, b)` will return a random integer between (and including) the two

integers `a` and `b` and `choice(a)` returns a random item from the list `a`. Look at these two lines of code, and figure out why they do the exact same thing:

```
random.randint(0, 9)
random.choice(range(0, 10))
```

Line 103 (line 109 in the new code) has also been changed. Now instead of returning the string `wordList[wordIndex]`, we are returning a list with two items. The first item is `wordDict[wordKey][wordIndex]`. The second item is `wordKey`. We return a list because we actually want the `getRandomWord()` to return two values, so putting those two values in a list and returning the list is the easiest way to do this.

`wordDict[wordKey][wordIndex]` may look kind of complicated, but it is just an expression you can evaluate one step at a time like anything else. First, imagine that `wordKey` had the value `'Fruits'` (which was chosen on line 104) and `wordIndex` has the value `5` (chosen on line 107). Here is how `wordDict[wordKey][wordIndex]` would evaluate:

```
wordDict[wordKey][wordIndex]

wordDict['Fruits'][5]

['apple', 'orange', 'lemon', 'lime', 'pear', 'watermelon',
'grape', 'grapefruit', 'cherry', 'banana', 'cantalope',
'mango', 'strawberry', 'tomato'][5]

'watermelon'
```

In the above case, the first item in the list this function returns would be the string `'watermelon'`.

There are just three more changes to make to our program. The first two are on the lines that we call the `getRandomWord()` function. The function is called on lines 148 and 184 in the original program:

```
147. correctLetters = ''
148. secretWord = getRandomWord(words)
149. gameIsDone = False

 ...

182.              gameIsDone = False
183.              secretWord = getRandomWord(words)
184.          else:
```

Because the `getRandomWord()` function now returns a list of two items instead of a string, `secretWord` will be assigned a list, not a string. We would then have to change the code as follows:

```
correctLetters = ''
secretWord = getRandomWord(words)
secretKey = secretWord[1]
secretWord = secretWord[0]
gameIsDone = False

 ...

            gameIsDone = False
            secretWord = getRandomWord(words)
            secretKey = secretWord[1]
            secretWord = secretWord[0]
        else:
```

With the above changes, `secretWord` is first a list of two items. Then we add a new variable named `secretKey` and set it to the second item in `secretWord`. Then we set `secretWord` itself to the first item in the `secretWord` list. That means that `secretWord` will then be a string.

But there is an easier way by doing a little trick with assignment statements. Try typing the following into the shell:

```
a, b, c = ['apples', 'cats', 42]
a
b
c
```

```
>>> a, b, c = ['apples', 'cats', 42]
>>> a
'apples'
>>> b
'cats'
>>> c
42
>>>
```

The trick is to put the same number of variables (delimited by commas) on the left side of the = sign as are in the list on the right side of the = sign. Python will automatically assign the first item's value in the list to the first variable, the second item's value to the second variable, and so on. But if you do not have

the same number of variables on the left side as there are items in the list on the right side, the Python interpreter will give you an error.

```
>>> a, b, c, d = ['apples', 'cats', 42]

Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    a, b, c, d = ['apples', 'cats', 42]
ValueError: need more than 3 values to unpack
>>> a, b, c = ['apples', 'cats']

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    a, b, c = ['apples', 'cats']
ValueError: need more than 2 values to unpack
>>> 
```

So we should change our code in Hangman to use this trick, which will mean our program uses fewer lines of code.

```
147. correctLetters = ''
148. secretWord, secretKey = getRandomWord(words)
149. gameIsDone = False

 ...

182.             gameIsDone = False
183.             secretWord, secretKey = getRandomWord
     (words)
184.         else:
```

The last change we will make is to add a simple print statement to tell the player which set of words they are trying to guess. This way, when the player plays the game they will know if the secret word is an animal, color, shape, or fruit. Add this line of code after line 151. Here is the original code:

```
151. while True:
152.     displayBoard(HANGMANPICS, missedLetters,
     correctLetters, secretWord)
```

Add the line so your program looks like this:

```
while True:
    print 'The secret word is in the set: ' + secretKey
    displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)
```

Now we are done with our changes. Instead of just a single list of words, the secret word will be chosen from many different lists of words. We will also tell the player which set of words the secret word is from. Try playing this new version. You can easily change the `words` dictionary on line 98 to include more sets of words.

We're done with Hangman. Let's move on to our next game, Tic Tac Toe!

Things Covered In This Chapter:

- Designing our game by drawing a flow chart before programming.
- ASCII Art
- Multi-line Strings
- Lists
- List indexes
- Index assignment
- List concatenation
- The `in` operator
- The `del` operator
- Methods
- The `append()` list method
- The `lower()` and `upper()` string methods
- The `reverse()` list method
- The `split()` list method
- The `len()` function
- Empty lists
- The `range()` function
- `for` loops
- Strings act like lists
- Mutable sequences (lists) and immutable sequences (strings)
- List slicing and substrings
- `elif` statements
- The `startswith(someString)` and `endswith(someString)` string

methods

- The dictionary data type (which is unordered, unlike list data type which is ordered)
- key-value pairs
- The `keys()` and `values()` dictionary methods
- Multiple variable assignment, such as `a, b, c = [1, 2, 3]`

Chapter 6 - Tic Tac Toe

We will now create a Tic Tac Toe game where the player plays against a simple artificial intelligence. An **artificial intelligence** (or **AI**) is a computer program that can intelligently respond to the player's moves. This game doesn't introduce any complicated new concepts. We will see that the artificial intelligence that plays Tic Tac Toe is really just several lines of code. So in a new file editor window, type in this source code and save it as tictactoe.py. Then run the game by pressing F5.

Sample Run

```
Welcome to Tic Tac Toe!
Do you want to be X or O?
X
The computer will go first.
   |   |
 O |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
What is your next move? (1-9)
3
   |   |
 O |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
 O |   | X
   |   |
What is your next move? (1-9)
4
   |   |
 O |   | O
   |   |
-----------
   |   |
```

```
 X |   |
   |   |
-----------
   |   |
 O |   | X
   |   |
What is your next move? (1-9)
5
   |   |
 O | O | O
   |   |
-----------
   |   |
 X | X |
   |   |
-----------
   |   |
 O |   | X
   |   |
The computer has beaten you! You lose.
Do you want to play again? (yes or no)
no
```

Source Code

tictactoe.py

```
# Tic Tac Toe

import random

def drawBoard(board):
    # This function prints out the board that it was passed.

    # "board" is a list of 10 strings representing the board (ignore index 0)
    print '   |   |'
    print ' ' + board[7] + ' | ' + board[8] + ' | ' + board[9]
    print '   |   |'
    print '-----------'
    print '   |   |'
    print ' ' + board[4] + ' | ' + board[5] + ' | ' + board[6]
```

```
    print '   |   |'
    print '-----------'
    print '   |   |'
    print ' ' + board[1] + ' | ' + board[2] + ' | ' + board[3]
    print '   |   |'

def inputPlayerLetter():
    # Let's the player type which letter they want to be.
    # Returns a list with the player's letter as the first item, and the computer's letter as the second.
    letter = ''
    while not (letter == 'X' or letter == 'O'):
        print 'Do you want to be X or O?'
        letter = raw_input().upper()

    # the first element in the tuple is the player's letter, the second is the computer's letter.
    if letter == 'X':
        return ['X', 'O']
    else:
        return ['O', 'X']

def whoGoesFirst():
    # Randomly choose the player who goes first.
    if random.randint(0, 1) == 0:
        return 'computer'
    else:
        return 'player'

def playAgain():
    # This function returns True if the player wants to play again, otherwise it returns False.
    print 'Do you want to play again? (yes or no)'
    return raw_input().lower().startswith('y')

def makeMove(board, letter, move):
    board[move] = letter

def isWinner(bo, le):
    # Given a board and a player's letter, this function returns True if that player has won.
    # We use bo instead of board and le instead of letter so we don't have to type as much.
    return ((bo[7] == le and bo[8] == le and bo[9] == le) or # across the top
```

```
    (bo[4] == le and bo[5] == le and bo[6] == le) or # across the middle
    (bo[1] == le and bo[2] == le and bo[3] == le) or # across the bottom
    (bo[7] == le and bo[4] == le and bo[1] == le) or # down the left side
    (bo[8] == le and bo[5] == le and bo[2] == le) or # down the middle
    (bo[9] == le and bo[6] == le and bo[3] == le) or # down the right side
    (bo[7] == le and bo[5] == le and bo[3] == le) or # diagonal
    (bo[9] == le and bo[5] == le and bo[1] == le)) # diagonal

def getBoardCopy(board):
    # Make a duplicate of the board list and return it the duplicate.
    dupeBoard = []

    for i in board:
        dupeBoard.append(i)

    return dupeBoard

def isSpaceFree(board, move):
    # Return true if the passed move is free on the passed board.
    return board[move] == ' '

def getPlayerMove(board):
    # Let the player type in their move.
    move = ' '
    while move not in '1 2 3 4 5 6 7 8 9'.split() or not isSpaceFree(board, int(move)):
        print 'What is your next move? (1-9)'
        move = raw_input()
    return int(move)

def chooseRandomMoveFromList(board, movesList):
    # Returns a valid move from the passed list on the passed board.
    # Returns None if there is no valid move.
    possibleMoves = []
    for i in movesList:
        if isSpaceFree(board, i):
            possibleMoves.append(i)
```

```

    if len(possibleMoves) != 0:
        return random.choice(possibleMoves)
    else:
        return None

def getComputerMove(board, computerLetter):
    # Given a board and the computer's letter, determine where to move and return that move.
    if computerLetter == 'X':
        playerLetter = 'O'
    else:
        playerLetter = 'X'

    # Here is our algorithm for our Tic Tac Toe AI:
    # First, check if we can win in the next move
    for i in range(1, 9):
        copy = getBoardCopy(board)
        if isSpaceFree(copy, i):
            makeMove(copy, computerLetter, i)
            if isWinner(copy, computerLetter):
                return i

    # Check if the player could win on their next move, and block them.
    for i in range(1, 9):
        copy = getBoardCopy(board)
        if isSpaceFree(copy, i):
            makeMove(copy, playerLetter, i)
            if isWinner(copy, playerLetter):
                return i

    # Try to take one of the corners, if they are free.
    move = chooseRandomMoveFromList(board, [1, 3, 7, 9])
    if move != None:
        return move

    # Try to take the center, if it is free.
    if isSpaceFree(board, 5):
        return 5

    # Move on one of the sides.
    return chooseRandomMoveFromList(board, [2, 4, 6, 8])

def isBoardFull(board):
```

```
    # Return True if every space on the board has been taken. Otherwise return False.
    for i in range(1, 10):
        if isSpaceFree(board, i):
            return False
    return True


print 'Welcome to Tic Tac Toe!'

while True:
    # Reset the board
    theBoard = [' '] * 10
    playerLetter, computerLetter = inputPlayerLetter()
    turn = whoGoesFirst()
    print 'The ' + turn + ' will go first.'
    gameIsPlaying = True

    while gameIsPlaying:
        if turn == 'player':
            # Player's turn.
            drawBoard(theBoard)
            move = getPlayerMove(theBoard)
            makeMove(theBoard, playerLetter, move)

            if isWinner(theBoard, playerLetter):
                drawBoard(theBoard)
                print 'Hooray! You have won the game!'
                gameIsPlaying = False
            else:
                if isBoardFull(theBoard):
                    drawBoard(theBoard)
                    print 'The game is a tie!'
                    break
                else:
                    turn = 'computer'

        else:
            # Computer's turn.
            move = getComputerMove(theBoard, computerLetter)
            makeMove(theBoard, computerLetter, move)

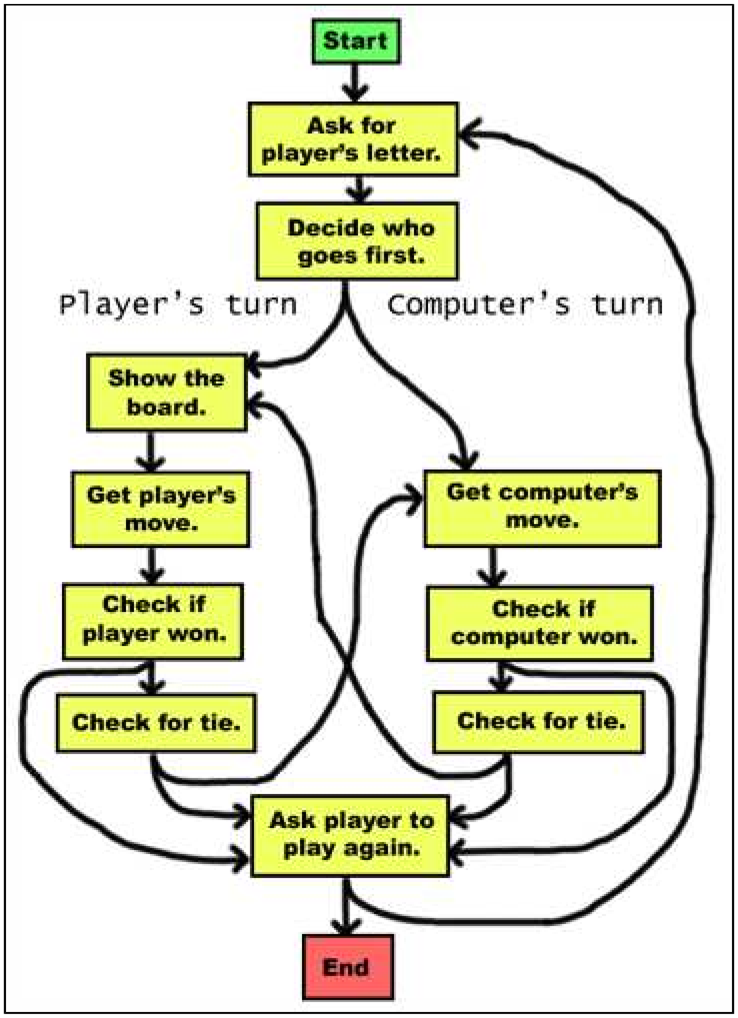
            if isWinner(theBoard, computerLetter):
                drawBoard(theBoard)
                print 'The computer has beaten you! You lose.'
```

```
                gameIsPlaying = False
            else:
                if isBoardFull(theBoard):
                    drawBoard(theBoard)
                    print 'The game is a tie!'
                    break
                else:
                    turn = 'player'

    if not playAgain():
        break
```

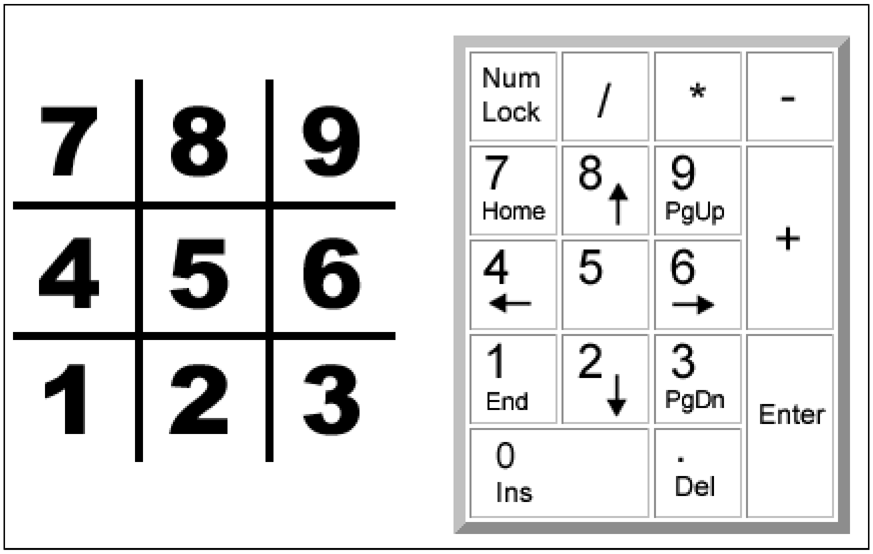
Designing the Program

Tic Tac Toe is a very easy and short game to play on paper. In our Tic Tac Toe computer game, we'll let the player choose if they want to be X or O, randomly choose who goes first, and then let the player and computer take turns making moves on the board. Here is what a flow chart of this game could look like:



You can see a lot of the boxes on the left side of the chart are what happens during the player's turn. The right side of the chart shows what happens on the computer's turn. The player has an extra box for drawing the board because the computer doesn't need the board printed on the screen. After the player or computer makes a move, we check if they won or caused a tie, and then the game switches turns. If either the computer or player ties or wins the game, we ask the player if they want to play again.

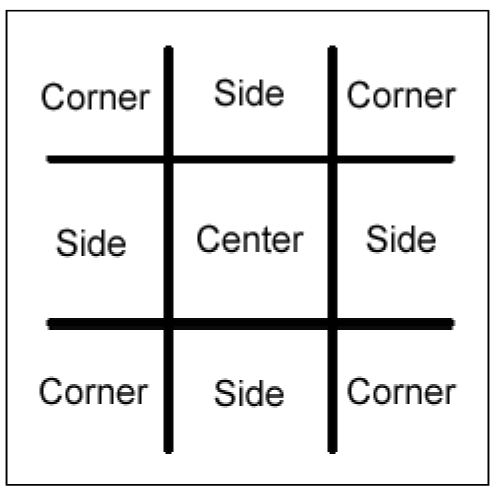
First, we need to figure out how we are going to represent the board as a variable. We are going to represent the Tic Tac Toe board as a list of ten strings. The ten strings will represent each of the nine positions on the board (and we will ignore one of our strings). The strings will either be `'X'` for the X player, `'O'` for the O player, or a space string `' '` to mark a spot on the board where no one has marked yet. To make it easier to remember which index in the list is for which piece, we will mirror the numbers on the keypad of our keyboard. (Because there is no 0 on the keypad, we will just ignore the string at index 0 in our list.)



So if we had a list with ten strings named `board`, then `board[7]` would be the top-left square on the board (either an X, O, or blank space). `board[5]` would be the very center. When the player types in which place they want to move, they will type a number from 1 to 9.

Game AI

Just to be clear, we will label three types of spaces on the Tic Tac Toe board: corners, sides, and the center. Here is a chart of what each space is:

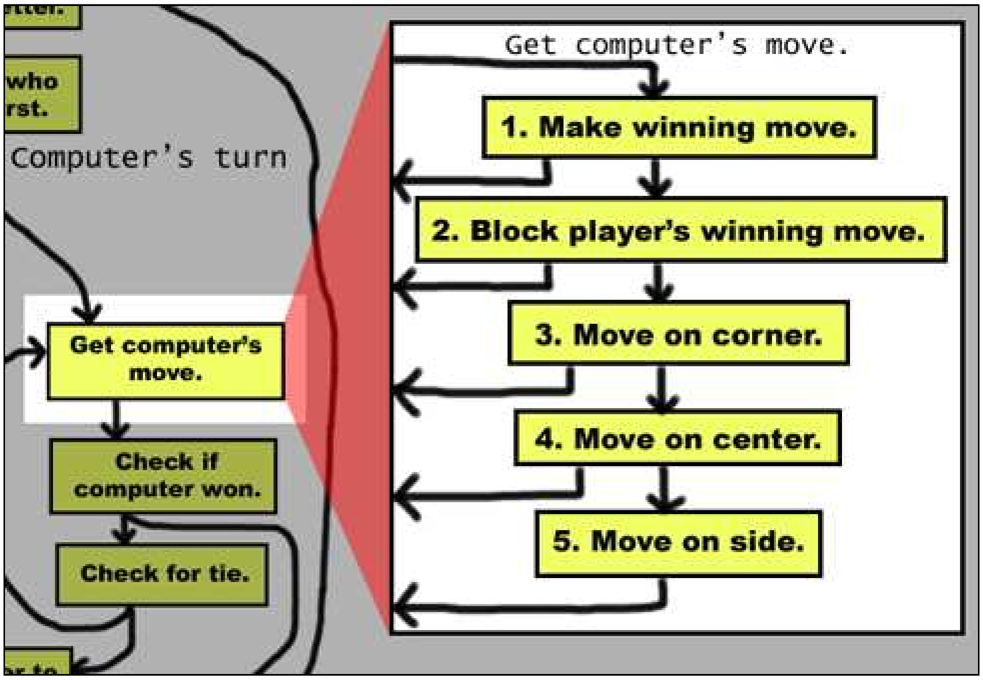


The AI for this game will follow a simple algorithm. An **algorithm** is a series of instructions to compute something. Our Tic Tac Toe AI's algorithm will determine which is the best place to move. Our algorithm will have the following steps:

1. First, see if there is a move the computer can make that will win the game. If there is, take that move. Otherwise, go to step 2.
2. See if there is a move the player can make that will cause the computer to lose the game. If there

is, we should move there to block the player. Otherwise, go to step 3.
3. Check if any of the corner spaces (spaces 1, 3, 7, or 9) are free. (We always want to take a corner piece instead of the center or a side piece.) If no corner piece is free, then go to step 4.
4. Check if the center is free. If so, move there. If it isn't, then go to step 5.
5. Move on any of the side pieces (spaces 2, 4, 6, or 8). There are no more steps, because if we have reached step 5 the side spaces are the only spaces left.

This all takes place in the "Get computer's move." box on our flow chart. We could add this information to our flow chart like this:



We will implement this algorithm as code in our `getComputerMove()` function.

Code Explanation

```
1. # Tic Tac Toe
2.
3. import random
```

A comment and importing the `random` module so we can use the `randint()` function in our game.

```
5. def drawBoard(board):
```

```
    # This function prints out the board that it was passed.

    # "board" is a list of 10 strings representing the board (ignore index 0)
    print '   |   |'
    print ' ' + board[7] + ' | ' + board[8] + ' | ' + board[9]
    print '   |   |'
    print '-----------'
    print '   |   |'
    print ' ' + board[4] + ' | ' + board[5] + ' | ' + board[6]
    print '   |   |'
    print '-----------'
    print '   |   |'
    print ' ' + board[1] + ' | ' + board[2] + ' | ' + board[3]
    print '   |   |'
```

This function will print out the game board, marked as directed by the `board` parameter. Many of our functions will work by passing the board as a list of ten strings to our functions. Be sure to get the spacing right in the strings that are printed, otherwise the board will look funny when it is printed on the screen.

Just as an example, here are some values that the `board` parameter could have (on the left) and what the `drawBoard()` function would print out:

<table>
<tr><th>board data structure</th><th>drawBoard(board) output</th></tr>
<tr><td>[' ', ' ', ' ', ' ', 'X', 'O', ' ', 'X', ' ', 'O']</td><td><pre>
   | |
 X | | O
   | |

   | |
 X | O |
   | |

   | |
   | |
   | |
</pre></td></tr>
<tr><td></td><td><pre>
   | |
   | |
   | |
</pre></td></tr>
</table>

<table>
<tr><td>[' ', 'O', 'O', ' ', ' ', 'X', ' ', ' ', ' ', ' ']</td><td>-----------
   | |
   | X |
   | |

   | |
 O | O |
   | |</td></tr>
<tr><td>[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']</td><td> | |
   | |
   | |

   | |
   | |
   | |

   | |
   | |
   | |</td></tr>
<tr><td>[' ', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']</td><td> | |
 X | X | X
   | |

   | |
 X | X | X
   | |

   | |
 X | X | X
   | |</td></tr>
</table>

The last board filled with X's could not possibly have happened (unless the X player skipped all of the O player's turns!) But the `drawBoard()` function doesn't care. It just prints the `board` parameter that it was passed.

```
def inputPlayerLetter():
    # Let's the player type which letter they want to be.
    # Returns a list with the player's letter as the first item, and the computer's letter as the second.
    letter = ''
    while not (letter == 'X' or letter == 'O'):
        print 'Do you want to be X or O?'
```

```
27.         letter = raw_input().upper()
```

The `inputPlayerLetter()` is a simple function. It asks if the player wants to be X or O, and will keep asking the player (with the `while` loop) until the player types in an X or O. Notice on line 26 that we automatically change the string returned by the call to `raw_input()` to uppercase letters with the `upper()` string method.

The `while` loop's condition contains parentheses, which means the expression inside the parentheses is evaluated first. If the `letter` variable was set to `'X'`, the expression would evaluate like this:

```
while not (letter == 'X' or letter == 'O'):

while not ('X' == 'X' or 'X' == 'O'):

while not (True or False):

while not (True):

while not True:

while False:
```

As you can see, if `letter` has the value `'X'` or `'O'`, then the loop's condition will be `False` and lets the program execution continue.

```
29.     # the first element in the tuple is the player's
    letter, the second is the computer's letter.
30.     if letter == 'X':
31.         return ['X', 'O']
32.     else:
33.        return ['O', 'X']
```

This function returns a list with two items. The first item will be the player's letter, and the second will be the computer's letter. This if-else statement chooses the appropriate list to return. This is much like the `getRandomWord()` function in the extended version of our Hangman game in the last chapter.

```
def whoGoesFirst():
    # Randomly choose the player who goes first.
    if random.randint(0, 1) == 0:
        return 'computer'
    else:
        return 'player'
```

The `whoGoesFirst()` function does a coin flip to determine who goes first, the computer or the player. Instead of flipping an actual coin, this code gets a random number of either `0` or `1` by calling the `random.randint()` function. If this function call returns a `0`, the `whoGoesFirst()` function returns the string `'computer'`. Otherwise, the function returns the string `'player'`. The code that calls this function will use the return value to know who will make the first move of the game.

```
def playAgain():
    # This function returns True if the player wants to play again, otherwise it returns False.
    print 'Do you want to play again? (yes or no)'
    return raw_input().lower().startswith('y')
```

The `playAgain()` function asks the player if they want to play another game. The function returns `True` if the player types in `'yes'` or `'YES'` or `'y'` or anything that begins with the letter Y. For any other response, the function returns `False`. The order of the method calls on line 151 is important. The return value from the call to the `raw_input()` function is a string that has its `lower()` method called on it. The `lower()` method returns another string (the lowercase string) and that string has its `startswith()` method called on it, passing the argument `'y'`.

There is no loop, because we assume that if the user entered anything besides a string that begins with `'y'`, they want to stop playing. So, we only ask the player once.

```
def makeMove(board, letter, move):
    board[move] = letter
```

The `makeMove()` function is very simple and only one line. The parameters are a list with ten strings named `board`, one of the player's letters (either `'X'` or `'O'`) named `letter`, and a place on the board where that player wants to go (which is an integer from `1` to `9`) named `move`.

But wait a second. You might think that this function doesn't do much. It seems to change one of the items in the `board` list to the value in `letter`. But because this code is in a function, the `board` variable will be forgotten when we exit this function and leave the function's scope.

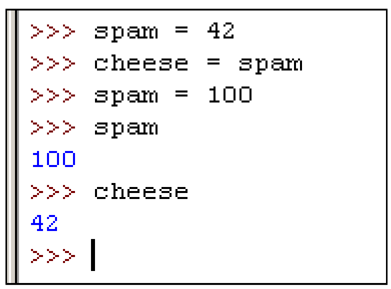
Actually, this is not the case. This is because lists are special when you pass them as arguments to functions. This is because you pass a reference to the list.

List References

Try typing the following into the shell:

```
spam = 42
cheese = spam
spam = 100
spam
cheese
```

When you type this into the shell, it should look like this:



This makes sense from what we know so far. We assign `42` to the `spam` variable, then we copy the value in `spam` and assign it to the variable `cheese`. When we later change the value in `spam` to `100`, this doesn't affect the value in `cheese`.

But lists don't work this way. When you assign a list to a variable with the = sign, you are actually assigning a reference to the list. A **reference** is a pointer to some bit of data. When you assign a list variable to a second variable, you are actually copying the reference and no the list itself. This is because the first variable doesn't contain a list; it contains a *reference* to a list.

Here is some code that will make this easier to understand. Type this into the shell:

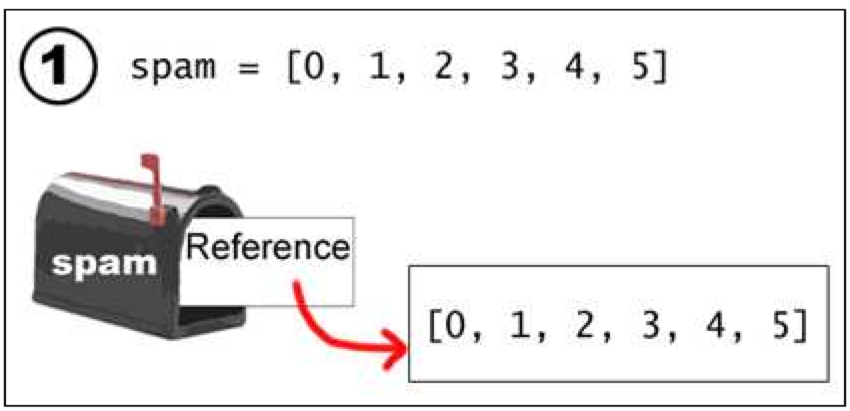
```
spam = [0, 1, 2, 3, 4, 5]
cheese = spam
cheese[1] = 'Hello!'
spam
cheese
```

This code will look like this:

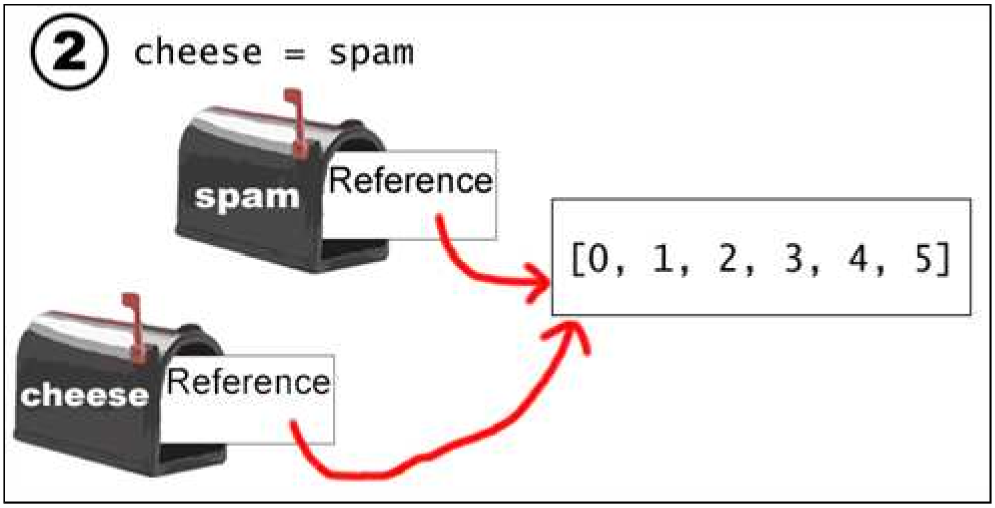
```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = spam
>>> cheese[1] = 'Hello!'
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
>>>
```

Notice that the line `cheese = spam` copies the reference in `spam` to `cheese`. This means that both `spam` and `cheese` refer to the same list. So when you modify `cheese` on the `cheese[1] = 'Hello!'` line, you are modifying the same list that `spam` refers to. This is why `spam` seems to have the same list value that `cheese` does.

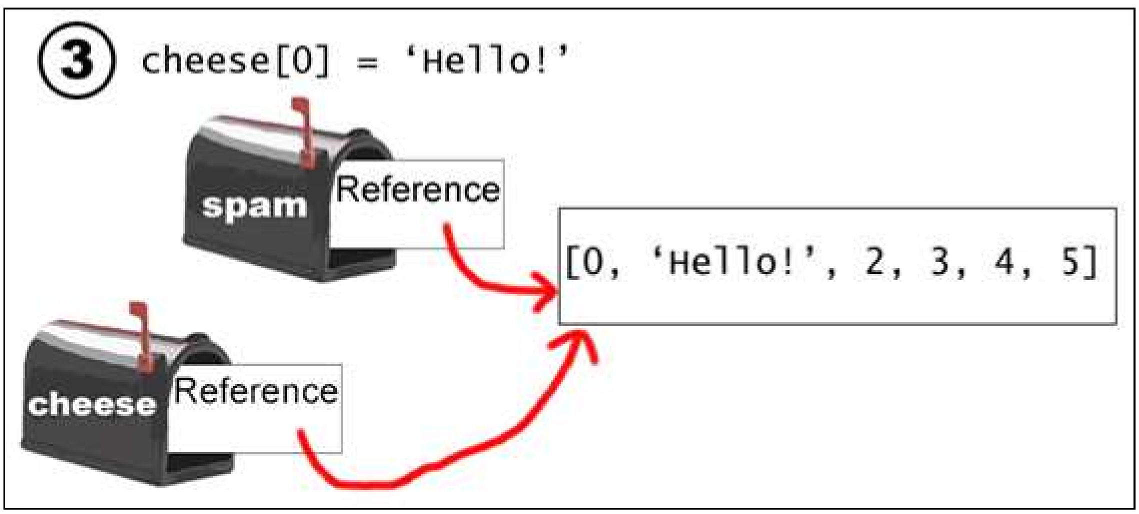
Remember when you first learned about variables, I said that variables were like mailboxes that contain values. List variables don't actually contain lists at all, they contain references to lists. Here are some pictures that explain what happens in the code you just typed in:



On the first line, the actual list is not contained in the `spam` variable but a reference to the list.



When you assign the reference in `spam` to `cheese`, the `cheese` variable contains a copy of the reference in `spam`. Now both `cheese` and `spam` refer to the same list.



When you alter the list that `cheese` refers to, the list that `spam` refers to is also changed because they are the same list.

Let's go back to the `makeMove()` function:

```
def makeMove(board, letter, move):
    board[move] = letter
```

When we pass a list value as the argument for the `board` parameter, we are actually passing a copy of the reference, not the list itself. The `letter` and `move` parameters are copies of the string and

integer values that we pass. Since they are copies, if we modify `letter` or `move` in this function, the original variables we used when we called `makeMove()` would not be modified. Only the copies would be modified.

But a copy of the reference still refers to the same list that the original reference refers to. So if we make changes to `board` in this function, the original list is modified. When we exit the `makeMove()` function, the copy of the reference is forgotten along with the other parameters. But since we were actually changing the original list, those changes remain after we exit the function. This is how the `makeMove()` function modifies the argument that it is passed.

```
50. def isWinner(bo, le):
51.     # Given a board and a player's letter, this function returns True if that player has won.
52.     # We use bo instead of board and le instead of letter so we don't have to type as much.
53.     return ((bo[7] == le and bo[8] == le and bo[9] == le) or # across the top
54.     (bo[4] == le and bo[5] == le and bo[6] == le) or # across the middle
55.     (bo[1] == le and bo[2] == le and bo[3] == le) or # across the bottom
56.     (bo[7] == le and bo[4] == le and bo[1] == le) or # down the left side
57.     (bo[8] == le and bo[5] == le and bo[2] == le) or # down the middle
58.     (bo[9] == le and bo[6] == le and bo[3] == le) or # down the right side
59.     (bo[7] == le and bo[5] == le and bo[3] == le) or # diagonal
60.     (bo[9] == le and bo[5] == le and bo[1] == le)) # diagonal
```

Lines 53 to 60 in the `isWinner()` function are actually one very long `if` statement. We use `bo` and `le` for the board and letter parameters so that we have less to type in this function. (This is a trick programmers sometimes use to reduce the amount they need to type. Be sure to add a comment though, otherwise you may forget what `bo` and `le` are supposed to mean.)

There are eight possible ways to win at Tic Tac Toe. First, have a line across the top, middle, and bottom. Second, have a line down the left, middle, or right. And finally, have either of the two diagonals. Note that each line of the condition checks if the three spaces are equal to the letter provided (combined with the `and` operator) and we use the `or` operator to combine the eight different ways to win. This means only one of the eight ways must be true in order for us to say that the player who owns letter in `le` is the winner.

Let's pretend that `le` is `'O'`, and the board looks like this:

```
   |   |
 X |   |
   |   |
-----------
   |   |
   | X |
   |   |
-----------
   |   |
 O | O | O
   |   |
```

If the board looks like that, then `bo` must be equal to `[' ', 'O', 'O', 'O', ' ', 'X', ' ', 'X', ' ', ' ']`. Here is how the expression after the `return` keyword on line 53 would evaluate:

```
53.     return ((bo[7] == le and bo[8] == le and bo[9] ==
    le) or # across the top
54.     (bo[4] == le and bo[5] == le and bo[6] == le) or #
    across the middle
55.     (bo[1] == le and bo[2] == le and bo[3] == le) or #
    across the bottom
56.     (bo[7] == le and bo[4] == le and bo[1] == le) or #
    down the left side
57.     (bo[8] == le and bo[5] == le and bo[2] == le) or #
    down the middle
58.     (bo[9] == le and bo[6] == le and bo[3] == le) or #
    down the right side
59.     (bo[7] == le and bo[5] == le and bo[3] == le) or #
    diagonal
60.     (bo[9] == le and bo[5] == le and bo[1] == le)) #
    diagonal
```



```
53.     return (('X' == 'O' and ' ' == 'O' and ' ' == 'O')
    or # across the top
54.     (' ' == 'O' and 'X' == 'O' and ' ' == 'O') or #
    across the middle
55.     ('O' == 'O' and 'O' == 'O' and 'O' == 'O') or #
    across the bottom
56.     ('X' == 'O' and ' ' == 'O' and 'O' == 'O') or # down
```

```
the left side
    (' ' == 'O' and 'X' == 'O' and 'O' == 'O') or # down the middle
    (' ' == 'O' and ' ' == 'O' and 'O' == 'O') or # down the right side
    ('X' == 'O' and 'X' == 'O' and 'O' == 'O') or # diagonal
    (' ' == 'O' and 'X' == 'O' and 'O' == 'O')) # diagonal
```

```
    return ((False and False and False) or # across the top
    (False and False and False) or # across the middle
    (True and True and True) or # across the bottom
    (False and False and True) or # down the left side
    (False and False and True) or # down the middle
    (False and False and True) or # down the right side
    (False and False and True) or # diagonal
    (False and False and True)) # diagonal
```

```
    return ((False) or # across the top
    (False) or # across the middle
    (True) or # across the bottom
    (False) or # down the left side
    (False) or # down the middle
    (False) or # down the right side
    (False) or # diagonal
    (False)) # diagonal
```

```
    return (False or # across the top
    False or # across the middle
    True or # across the bottom
    False or # down the left side
    False or # down the middle
    False or # down the right side
    False or # diagonal
    False) # diagonal
```

```
    return (True)
```

```
    return True
```

So given those values for `bo` and `le`, the expression would evaluate to `True`. Remember that the value of `le` matters. If `le` is `'O'` and X has won the game, the `isWinner()` would return `False`.

```
62. def getBoardCopy(board):
63.     # Make a duplicate of the board list and return it the duplicate.
64.     dupeBoard = []
65.
66.     for i in board:
67.         dupeBoard.append(i)
68.
69.     return dupeBoard
```

The `getBoardCopy()` function is here so that we can easily make a copy of a given 10-string list that represents a Tic Tac Toe board in our game. There are times that we will want our AI algorithm to make temporary modifications to the board without changing the original board. In that case, we call this function.

Line 64 actually creates a brand new board, because it is not copying another variable's reference to an existing board. The `for` loop will go through the `board` argument that is passed to this function, appending the values in the original board to our duplicate board. Finally, after the loop, we will return the `dupeBoard` variable's reference to the duplicate board.

```
def isSpaceFree(board, move):
    # Return true if the passed move is free on the passed board.
    return board[move] == ' '
```

This is a simple function that, given a Tic Tac Toe board and a possible move, will return if that move is available or not. Remember that free spaces on our board lists are marked as a single space string.

```
75. def getPlayerMove(board):
76.     # Let the player type in their move.
77.     move = ' '
78.     while move not in '1 2 3 4 5 6 7 8 9'.split() or
    not isSpaceFree(board, int(move)):
79.         print 'What is your next move? (1-9)'
80.         move = raw_input()
81.     return int(move)
```

The `getPlayerMove()` function asks the player to enter the number for the space they wish to move. The function makes sure that they enter a space that is 1) a valid space (an integer 1 through 9), and 2) a space that is not already taken, given the Tic Tac Toe board passed to the function in the `board` parameter.

The two lines of code inside the `while` loop simply ask the player to enter a number from 1 to 9. The loop's condition will keep looping, that is, it will keep asking the player for a space, as long as the condition is `True`. The condition is `True` if either of the expressions on the left or right side of the `or` keyword is `True`.

The expression on the left side checks if the move that the player entered is equal to `'1'`, `'2'`, `'3'`, and so on up to `'9'` by creating a list with these strings (with the `split()` method) and checking if `move` is in this list. `'1 2 3 4 5 6 7 8 9'.split()` evaluates to be the same as `['1', '2', '3', '4', '5', '6', '7', '8', '9']`, but it easier to type.

The expression on the right side checks if the move that the player entered is a free space on the board. It checks this by calling the `isSpaceFree()` function we just wrote. Remember that `isSpaceFree()` will return `True` if the move we pass is available on the board. Note that `isSpaceFree()` expects an integer for `move`, so we use the `int()` function to evaluate an integer form of `move`.

We add the `not` operators to both sides so that the condition will be `True` when both of these requirements are unfulfilled. This will cause the loop to ask the player again and again until they enter a proper move.

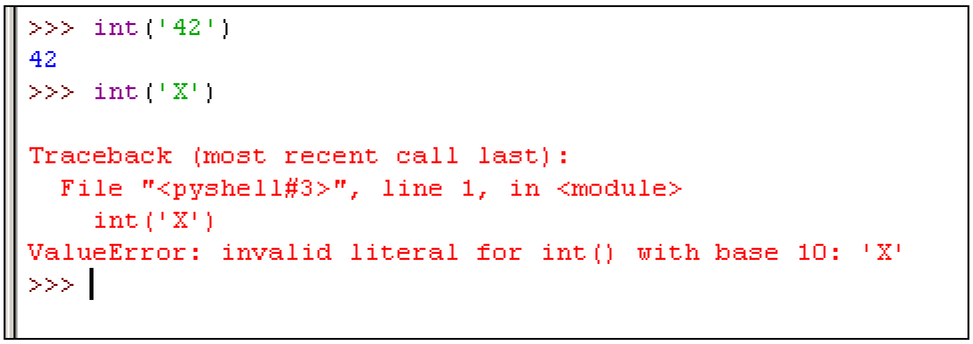
Finally, on line 81, we will return the integer form of whatever move the player entered. Remember that `raw_input()` returns a string, so we will want to use the `int()` function to evaluate the string as an integer.

Short-Circuit Evaluation

You may have noticed there is a possible problem in our `getPlayerMove()` function. What if the player typed in `'X'` or some other non-integer string? The `move not in '1 2 3 4 5 6 7 8 9'.split()` would return `False` as expected, and then we would evaluate the expression on the right side. But when we pass `'X'` (the value in `move` to the `int()` function, the call to `int()` would give us an error. It gives us this error because the `int()` function can only take strings of number characters, like `'9'` or `'42'`, not strings like `'X'`

As an example of this kind of error, try typing this into the shell:

```
int('42')
int('X')
```



But when you play our Tic Tac Toe game and try typing `'X'` in for your move, this error doesn't happen. The reason is because the `while` loop's condition is being short-circuited.

What short-circuiting means is that because the expression on the left side of the `or` keyword (`move not in '1 2 3 4 5 6 7 8 9'.split()`) evaluates to `True`, the Python interpreter knows that the entire expression will evaluate to `True` no matter if the expression on the right side of the `or` keyword evaluates to `True` or `False`. Python doesn't even bother calling the function. This works out well for us, because if the expression on the right side is `True` then `move` is not a string in number form. That would cause `int()` to give us an error. The only times `move not in '1 2 3 4 5 6 7 8 9'.split()` evaluates to `False` are when `move` is not a single-digit string. In that case, the call to `int()` would not give us an error.

Here's a short program that gives a good example of short-circuiting. Open a new file in the IDLE editor and type in this program, then press F5 to run it:

```
def TrueFizz(message):
    print message
    return True

def FalseFizz(message):
    print message
    return False
```

```
 8.
 9. if FalseFizz('Cats') or TrueFizz('Dogs'):
10.     print 'Step 1'
11.
12. if TrueFizz('Hello') or TrueFizz('Goodbye'):
13.     print 'Step 2'
14.
15. if TrueFizz('Spam') and TrueFizz('Cheese'):
16.     print 'Step 3'
17.
18. if FalseFizz('Red') and TrueFizz('Blue'):
19.     print 'Step 4'
```

When you run this program, you can see the output:

```
Cats
Dogs
Step 1
Hello
Step 2
Spam
Cheese
Step 3
Red
```

This small program has two functions: `TrueFizz()` and `FalseFizz()`. `TrueFizz()` will display a message and return the value `True`, while `FalseFizz()` will display a message and return the value `False`. This will help us determine when these functions are being called, or when these functions are being skipped due to short-circuiting.

The first `if` statement on line 9 in our small program will first evaluate `TrueFizz()`. We know this happens because `Cats` is printed to the screen. The entire expression could still be `True` if the expression to the right of the `or` keyword is `True`. So the call `TrueFizz('Dogs')` is evaluated, `Dogs` is printed to the screen and `True` is returned. The `if` statement's condition evaluates to `False or True`, which in turn evaluates to `True`. `Step 1` is then printed to the screen. No short-circuiting took place for this expression's evaluation.

The second `if` statement on line 12 does have short-circuiting. This is because when we call `TrueFizz('Hello')`, it prints `Hello` and returns `True`. Because it doesn't matter what is on the right side of the `or` keyword, the Python interpreter doesn't call `TrueFizz('Goodbye')`. You can tell it is not called because `Goodbye` is not printed to the screen. The `if` statement's condition is `True`, so `Step 2` is printed to the screen.

The third `if` statement on line 15 does not have short-circuiting. The call to `TrueFizz('Spam')` returns `True`, but we do not know if the entire condition is `True` or `False` because of the `and`

operator. So Python will call `TrueFizz('Cheese')`, which prints `Cheese` and returns `True`. The `if` statement's condition is evaluated to `True and True`, which in turn evaluates to `True`. Because the condition is `True`, `Step 3` is printed to the screen.

The fourth `if` statement on line 18 does have short-circuiting. The `FalseFizz('Red')` call prints `Red` and returns `False`. Because the left side of the `and` keyword is `False`, it does not matter if the right side is `True` or `False`, the condition will evaluate to `False` anyway. So `TrueFizz('Blue')` is not called and `Blue` does not appear on the screen. Because the `if` statement's condition evaluated to `False`, `Step 4` is not printed to the screen.

Short-circuiting can happen for any expression that includes the boolean operators, `and` or `or`. It is important to remember that this can happen; otherwise you may find that some function calls in the expression are never called and you will not know why.

Code Explanation continued...

```
def chooseRandomMoveFromList(board, movesList):
    # Returns a valid move from the passed list on the passed board.
    # Returns None if there is no valid move.
    possibleMoves = []
    for i in movesList:
        if isSpaceFree(board, i):
            possibleMoves.append(i)
```

The `chooseRandomMoveFromList` function will be of use to us when we are implementing the code for our AI. The first parameter `board` is the 10-string list that represents a Tic Tac Toe board. The second parameter `movesList` is a list of integers that represent possible moves. For example, if `movesList` is `[1, 3, 7, 9]`, then that means we should return the number for one of the corner spaces on the board. This function will choose one of those moves from the list. It also makes sure that the move that it chooses is not already taken. To do this, we create a blank list and assign it to `possibleMoves`. The `for` loop will go through the list of moves passed to this function in `movesList`. If that move is available (which we figure out with a call to `isSpaceFree()`), then we add it to `possibleMoves` with the `append()` method.

```
    if len(possibleMoves) != 0:
        return random.choice(possibleMoves)
    else:
        return None
```

At this point, the `possibleMoves` list has all of the moves that were in `movesList` that are also free spaces on the board represented by `board`. If the list is not empty, then there is at least one possible move that can be made on the board.

This list might be empty. For example, if `movesList` was `[1, 3, 7, 9]` but the board represented by the `board` parameter had all the corner spaces already taken, the `possibleMoves` list would have been empty.

If `possibleMoves` is empty, then `len(possibleMoves)` will evaluate to `0` and the code in the else-block will execute. Notice that it returns something called `None`.

The None Value

None is a special value that you can assign to a variable. `None` is the only value of the data type `NoneType`. The `None` value represents the lack of a value. It can be very useful to use the `None` value when you have not set a variables value yet. For example, say you had a variable named `quizAnswer` which holds the user's answer to some True-False pop quiz question. You could set `quizAnswer` to `None` if the user skipped the question or did not answer it. Using `None` would be better because if you set it to `True` or `False` before assigning the value of the user's answer, it may look like the user gave an answer the question even though they didn't.

Calls to functions that do not return anything (that is, they exit by reaching the end of the function and not from a `return` statement) will evaluate to `None`. The `None` value is written without quotes and with a capital "N" and lowercase "one".

Code Explanation continued...

```
def getComputerMove(board, computerLetter):
    # Given a board and the computer's letter, determine where to move and return that move.
    if computerLetter == 'X':
        playerLetter = 'O'
    else:
        playerLetter = 'X'
```

The `None` function is where our AI will be coded. The parameters are a Tic Tac Toe board (in the `board` parameter) and which letter the computer is (either `'X'` or `'O'`). The first few lines simply assign the other letter to a variable named `None`. This lets us use the same code, no matter who is X and who is O. This function will return the integer that represents which space the computer will move.

Remember how our algorithm works:

1. First, see if there is a move the computer can make that will win the game. If there is, take that move. Otherwise, go to step 2.
2. See if there is a move the player can make that will cause the computer to lose the game. If there is, we should move there to block the player. Otherwise, go to step 3.
3. Check if any of the corner spaces (spaces 1, 3, 7, or 9) are free. (We always want to take a corner piece instead of the center or a side piece.) If no corner piece is free, then go to step 5.
4. Check if the center is free. If so, move there. If it isn't, then go to step 6.
5. Move on any of the side pieces (spaces 2, 4, 6, or 8). There are no more steps, because if we have reached step 6 the side spaces are the only spaces left.

```
103.     # Here is our algorithm for our Tic Tac Toe AI:
104.     # First, check if we can win in the next move
105.     for i in range(1, 9):
106.         copy = getBoardCopy(board)
107.         if isSpaceFree(copy, i):
108.             makeMove(copy, computerLetter, i)
109.             if isWinner(copy, computerLetter):
110.                 return i
```

More than anything, if the computer can win in one more move, the computer should make that move. We will do this by trying each of the nine spaces on the board with a `for` loop. The first line in the loop makes a copy of the `board` list. This is because we want to make a move on the copy of the board, and then see if that move results in the computer winning. We don't want to modify the original Tic Tac Toe board, which is why we make a call to `getBoardCopy()`. We check if the space we will move is free, and if so, we move on that space and see if this results in winning. If it does, we return that space's integer.

If moving on none of the spaces results in winning, then the loop will finally end and we move on to line 112.

```
112.     # Check if the player could win on their next
     move, and block them.
113.     for i in range(1, 9):
114.         copy = getBoardCopy(board)
115.         if isSpaceFree(copy, i):
116.             makeMove(copy, playerLetter, i)
117.             if isWinner(copy, playerLetter):
118.                 return i
```

At this point, we know we cannot win in one move. So we want to make sure the human player cannot win in one more move. The code is very similar, except on the copy of the board, we place the player's letter before calling the `isWinner()` function. If there is a position the player can move that will let them win, the computer should move there.

If the human player cannot win in one more move, the `for` loop will eventually stop and execution continues on to line 120.

```
120.     # Try to take one of the corners, if they are free.
121.     move = chooseRandomMoveFromList(board, [1, 3, 7, 9])
122.     if move != None:
123.         return move
```

Our call to `chooseRandomMoveFromList()` with the list of `[1, 3, 7, 9]` will ensure that it returns the integer for one of the corner spaces. (Remember, the corner spaces are represented by the integers `1`, `3`, `7`, and `9`.) If all the corner spaces are taken, our `chooseRandomMoveFromList()` function will return the `None` value. In that case, we will move on to line 125.

```
125.     # Try to take the center, if it is free.
126.     if isSpaceFree(board, 5):
127.         return 5
```

If none of the corners are available, we will try to move on the center space if it is free. If the center space is not free, the execution moves on to line 129.

```
129.     # Move on one of the sides.
130.     return chooseRandomMoveFromList(board, [2, 4, 6, 8])
```

This code also makes a call to `chooseRandomMoveFromList()`, except we pass it a list of the side spaces (`[2, 4, 6, 8]`). We know that this function will not return `None`, because the side spaces are the only spaces we have not yet checked. This is the end of the `getComputerMove()` function and our AI algorithm.

```
def isBoardFull(board):
    # Return True if every space on the board has been taken. Otherwise return False.
    for i in range(1, 10):
        if isSpaceFree(board, i):
            return False
    return True
```

The last function we will write is `isBoardFull()`, which returns `True` if the 10-string list `board` argument it was passed has an `'X'` or `'O'` on every single space. If there is at least one space in `board` that is set to a single space `' '` then it will return `False`.

The `for` loop will let us check spaces 1 through 9 on the Tic Tac Toe board. (Remember that `range(1, 10)` call returns the list `[1, 2, 3, 4, 5, 6, 7, 8, 9]`.) As soon as it finds a free space in the board (that is, when `isSpaceFree(board, i)` returns `True`), the `isBoardFull()` function will return `False`.

If execution manages to go through every iteration of the loop, we will know that none of the spaces are free. So at that point, we will execute `return True`.

```
140. print 'Welcome to Tic Tac Toe!'
```

Line 140 is the first line that isn't inside of a function, so it is the first line of code that is executed when we run this program.

```
142. while True:
143.     # Reset the board
144.     theBoard = [' '] * 10
```

This `while` loop has `True` for the condition, so that means we will keep looping in this loop until we encounter a `break` statement. Line 144 sets up the main Tic Tac Toe board that we will use, named `theBoard`. It is a 10-string list, where each string is a single space `' '`. Remember the little trick using the multiplication operator with a list to reproduce it. `[' '] * 10`. It evaluates to `[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']`, but is shorter for us to type.

```
145.     playerLetter, computerLetter = inputPlayerLetter()
```

The `inputPlayerLetter()` function lets the player type in whether they want to be X or O. The function returns a 2-string list, either `['X', 'O']` or `['O', 'X']`. We use the multiple assignment trick here that we learned in the Hangman chapter. If `inputPlayerLetter()` returns `['X', 'O']`, then `playerLetter` is `'X'` and `computerLetter` is `'O'`. If `inputPlayerLetter()` returns `['O', 'X']`, then `playerLetter` is `'O'` and `computerLetter` is `'X'`.

```
146.     turn = whoGoesFirst()
147.     print 'The ' + turn + ' will go first.'
148.     gameIsPlaying = True
```

The `whoGoesFirst()` function randomly decides who goes first, and returns either the string `'player'` or the string `'computer'`. On line 147, we tell the player who will go first. The `gameIsPlayer` variable is what we will use to keep track of whether the game has been won, lost, tied, or if is the other player's turn.

```
150.     while gameIsPlaying:
```

This is a loop that will keep going back and forth between the player's turn and the computer's turn, as long as `gameIsPlaying` is set to `True`.

```
151.         if turn == 'player':
152.             # Player's turn.
153.             drawBoard(theBoard)
154.             move = getPlayerMove(theBoard)
155.             makeMove(theBoard, playerLetter, move)
```

The `turn` variable was originally set by `whoGoesFirst()`. It is either set to `'player'` or `'computer'`. If `turn` contains the string `'computer'`, then the condition is `False` and execution will jump down to line 169.

The first thing we do when it is the player's turn (according to the flow chart we drew at the beginning of this chapter) is show the board to the player. The `drawBoard()` function, called with the `theBoard` variable, will print the board on the screen. We then let the player type in their move by calling our `getPlayerMove()` function, and set the move on the board by calling our `makeMove()` function.

```
157.             if isWinner(theBoard, playerLetter):
158.                 drawBoard(theBoard)
159.                 print 'Hooray! You have won the game!'
160.                 gameIsPlaying = False
```

Now that the player has made his move, our program should check if they have won the game with this move. If the `isWinner()` function returns `True`, we should show them the winning board (the previous call to `drawBoard()` shows the board BEFORE they made the winning move) and print a message telling them they have won.

Then we set `gameIsPlaying` to `False` so that execution does not continue on to the computer's turn.

```
161.             else:
162.                 if isBoardFull(theBoard):
163.                     drawBoard(theBoard)
164.                     print 'The game is a tie!'
165.                     break
```

If the player did not win with his last move, then maybe his last move filled up the entire board and we now have a tie. In this else-block, we check if the board is full with a call to the `isBoardFull()` function. If it returns `True`, then we should draw the board by calling `drawBoard()` and tell the player a tie has occurred. The `break` statement will break us out of the `while` loop we are in and jump down to line 186.

```
167.                 else:
168.                     turn = 'computer'
```

If the player has not won or tied the game, then we should just set the `turn` variable to

'computer' so that when this while loop loops back to the start it will execute the code for the computer's turn.

```
        else:
```

If the turn variable was not set to 'player', then we know it is the computer's turn and the code in this else-block will execute. This code is very similar to the code for the player's turn, except the computer does not need the board printed on the screen so we skip the call to the drawBoard() function.

```
            # Computer's turn.
            move = getComputerMove(theBoard, computerLetter)
            makeMove(theBoard, computerLetter, move)
```

This code is almost identical to the code for the player's turn.

```
            if isWinner(theBoard, computerLetter):
                drawBoard(theBoard)
                print 'The computer has beaten you! You lose.'
                gameIsPlaying = False
```

We want to check if the computer won with its last move. The reason we call drawBoard() here is because the player will want to see what move the computer made to win the game. We then set gameIsPlaying to False so that the game does not continue.

```
            else:
                if isBoardFull(theBoard):
                    drawBoard(theBoard)
                    print 'The game is a tie!'
                    break
```

These lines of code are identical to the code on lines 162 to 165. The only difference is this is a check for a tied game after the computer has moved.

```
183.                    else:
184.                        turn = 'player'
```

If the game is neither won nor tied, it then becomes the player's turn. There are no more lines of code inside the `while` loop, so execution would jump back to the `while` statement on line 150.

```
186.        if not playAgain():
187.            break
```

These lines of code are located immediately after the while-block started by the `while` statement on line 150. Remember, we would only exit out of that `while` loop if it's condition (the `gameIsPlaying` variable) was `False`. `gameIsPlaying` is set to `False` when the game has ended, so at this point we are going to ask the player if they want to play again.

Remember, when we evaluate the condition in this `False` statement, we call the `False` function which will let the user type in if they want to play or not. `playAgain()` will return `True` if the player typed something that began with a `'y'` like `'yes'` or `'y'`. Otherwise `playAgain()` will return `False`.

If `playAgain()` returns `False`, then the `if` statement's condition is `True` (because of the `not` operator that reverses the boolean value) and we execute the `break` statement. That breaks us out of the `while` loop that was started on line 142. But there are no more lines of code after that while-block, so the program terminates.

A Web Page for Program Tracing

And that's the entire Tic Tac Toe game. If you want to see the code in action, go to the following web page to see the how the program executes line by line:

- Tic Tac Toe, trace 1 - http://pythonbook.coffeeghost.net/trace1TicTacToe.html

Things Covered In This Chapter:

- Artificial Intelligence

- List References
- Short-Circuit Evaluation
- The None Value

Chapter 7 - Bagels

Bagels is a simple game you can play with a friend. Your friend thinks up a random 3-digit number, and you try to guess what the number is. After each guess, your friend gives you clues. If the friend tells you "bagels", that means that none of the three digits is in the secret number. If your friend tells you "pico", then one of the digits is in the secret number, but your guess has the digit in the wrong place. If your friend tells you "fermi", then your guess has a correct digit in the correct place. Of course, even if you get a pico or fermi clue, you still don't know which digit in your guess is the correct one.

You can also get multiple clues after each guess. Say the secret number is 456, and your guess is 546. The clue you get would be "fermi pico pico" because one digit is correct and in the correct place (the digit 6), and two digits are in the secret number but in the wrong place (the digits 4 and 5).

Sample Run

```
I am thinking of a 3-digit number. Try to guess what it is.
Here are some clues:
When I say:    That means:
  Pico         One digit is correct but in the wrong
position.
  Fermi        One digit is correct and in the right
position.
  Bagels       No digit is correct.
I have thought up a number. You have 10 guesses to get it.
Guess #1:
123
Fermi
Guess #2:
453
Pico
Guess #3:
425
Fermi
Guess #4:
326
Bagels
Guess #5:
489
Bagels
Guess #6:
075
Fermi Fermi
Guess #7:
015
Fermi Pico
```

```
Guess #8:
175
You got it!
Do you want to play again? (yes or no)
no
```

Source Code

bagels.py

```python
import random

def getSecretNum(numDigits):
    # Returns a string that is numDigits long, made up of random digits.
    secretNum = ''
    for i in range(numDigits):
        secretNum += random.choice('0 1 2 3 4 5 6 7 8 9'.split())

    return secretNum

def getClues(guess, secretNum):
    # Returns a string with the pico, fermi, bagels clues to the user.
    if guess == secretNum:
        return 'You got it!'

    clue = []

    for i in range(len(guess)):
        if guess[i] == secretNum[i]:
            clue.append('Fermi')
        elif guess[i] in secretNum:
            clue.append('Pico')
    if len(clue) == 0:
        return 'Bagels'

    clue.sort()
    return ' '.join(clue)

def isOnlyDigits(num):
    # Returns True if num is a string made up only of digits. Otherwise returns False.
    if num == '':
```

```
        return False

    for i in num:
        if i not in '0 1 2 3 4 5 6 7 8 9'.split():
            return False

    return True

def playAgain():
    # This function returns True if the player wants to play again, otherwise it returns False.
    print 'Do you want to play again? (yes or no)'
    return raw_input().lower().startswith('y')

NUMDIGITS = 3
MAXGUESS = 10

print 'I am thinking of a %s-digit number. Try to guess what it is.' % (NUMDIGITS)
print 'Here are some clues:'
print 'When I say:    That means:'
print '  Pico         One digit is correct but in the wrong position.'
print '  Fermi        One digit is correct and in the right position.'
print '  Bagels       No digit is correct.'

while True:
    secretNum = getSecretNum(NUMDIGITS)
    print 'I have thought up a number. You have %s guesses to get it.' % (MAXGUESS)

    numGuesses = 1
    while numGuesses <= MAXGUESS:
        guess = ''
        while len(guess) != NUMDIGITS or not isOnlyDigits(guess):
            print 'Guess #%s: ' % (numGuesses)
            guess = raw_input()

        clue = getClues(guess, secretNum)
        print clue
        numGuesses += 1

        if guess == secretNum:
            break
        if numGuesses > MAXGUESS:
```

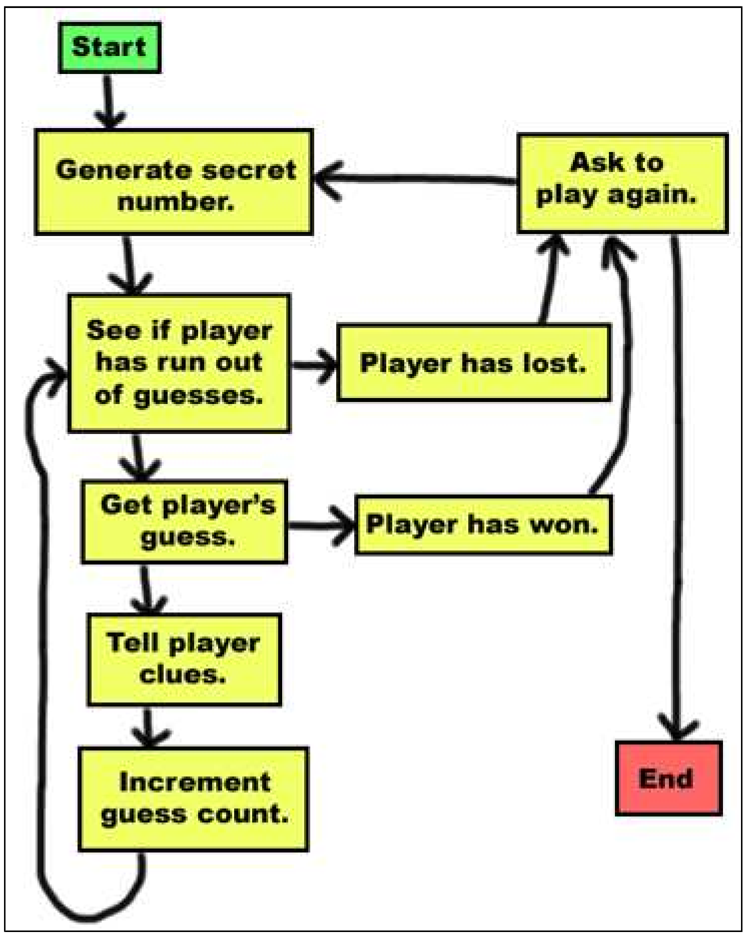
```
            print 'You ran out of guesses. The answer was %s.' % (secretNum)

    if not playAgain():
        break

```

Designing the Program

Here is a flow chart for this program. The flow chart describes the basic events of what happens in this game, and in what order they can happen:



And here is the source code for our game. Start a new file and type the code in, and then save the file as bagels.py. We will design our game so that it is very easy to change the size of the secret number. It can be 3 digits or 5 digits or 30 digits. We will do this by using a constant variable named `NUMDIGITS` instead of hard-coding the integer 3 into our source code.

Hard-coding means writing a program in a way that it changing the behavior of the program requires

changing a lot of the source code. For example, we could hard-code a name into a `print` statement like: `print 'Hello, Albert'`. Or we could use this line: `print 'Hello, ' + name` which would let us change the name that is printed by changing the `name` variable while the program is running.

Code Explanation

```
import random
```

This game imports the `random` module so we can use the module's random numbers function.

```
def getSecretNum(numDigits):
    # Returns a string that is numDigits long, made up of random digits.
    secretNum = ''
    for i in range(numDigits):
        secretNum += random.choice('0 1 2 3 4 5 6 7 8 9'.split())

    return secretNum
```

Our first function is named `getSecretNum()`, which will generate the random secret number. Instead of having the code only produce 3-digit numbers, we use a parameter named `numDigits` to tell us how many digits the secret number should have.

You may notice that `secretNum` in this function is a string, not an integer. This may seem odd, but remember that our secret number could be something like `'007'`. If we stored this as an integer, it would look like `7` which would make it harder to work with in our program.

This function is simple. The `secretNum` variable starts out as a blank string. We then loop a number of times equal to the integer value in `numDigits`. On each iteration through the loop, a new random digit is concatenated to the end of `secretNum`. So if `numDigits` is `3`, the loop will iterate three times and three random digits will be concatenated. (Remember, the `random.choice()` function returns a randomly chosen item from the list it is passed as an argument. `'0 1 2 3 4 5 6 7 8 9'` is a string, but the `split()` method called on it converts it into a list.)

Augmented Assignment Operators

The += operator is new. This is called an **augmented assignment operator**. Normally, if you wanted to add or concatenate a value to a variable, you would use code that looked like this:

```
spam = 42
spam = spam + 10

cheese = 'Hello '
cheese = cheese + 'world!'
```

After running the above code, spam would have the value 52 and cheese would have the value 'Hello world!'. The augmented assignment operators are a shortcut that frees you from retyping the variable name. The following code does the exact same thing as the above code:

```
spam = 42
spam += 10              # Same as spam = spam + 10

cheese = 'Hello '
cheese += 'world!'      # Same as cheese = cheese + 'world!'
```

There are other augmented assignment operators. -= will subtract a value from an integer. *= will multiply the variable by a value. /= will divide a variable by a value. Notice that these augmented assignment operators do the same math operations as the -, *, and / operators. Augmented assignment operators are a neat shortcut.

Code Explanation Continued...

```
def getClues(guess, secretNum):
    # Returns a string with the pico, fermi, bagels clues to the user.
    if guess == secretNum:
        return 'You got it!'
```

The getClues() function will return a string with the fermi, pico, and bagels clue, depending on what it is passed for the guess and secretNum parameters. The most obvious and easiest step is to check if the guess is the exact same as the secret number. In that case, we can just return 'You got it!'.

```
    clue = []

```

```
    for i in range(len(guess)):
        if guess[i] == secretNum[i]:
            clue.append('Fermi')
        elif guess[i] in secretNum:
            clue.append('Pico')
```

If the guess is not the exact same as the secret number, we need to figure out what clues to give the player. First we'll set up a list named `clue`, which we will add the strings `'Fermi'` and `'Pico'` as needed. We will combine the strings in this list into a single string to return.

We do this by looping through each possible index in `guess` and `secretNum`. We will assume that `guess` and `secretNum` are the same size (we can guarantee this in the code that calls `getClues()`.) The `if` statement checks if the first, second, third, etc. (depending on the value of `i` being `0`, `1`, `2`, etc.) letter of `guess` is the same as the number in the same position in `secretNum`. If so, we will add a string `'Fermi'` to `clue`.

If that condition is `False` we will check if the number at that position in `guess` exists in `secretNum`. If this condition is `True` we know that the number is somewhere in the secret number but not in the same position. This is why we add the `'Pico'` to `clue`.

```
    if len(clue) == 0:
        return 'Bagels'
```

If we go through the entire `for` loop above and never add anything to the `clue` list, then we know that there are no correct digits at all in `guess`. In this case, we should just return the string `'Bagels'` as our only clue.

The `sort()` List Method

```
    clue.sort()
```

Lists have a method named `sort()` that rearranges the items in the list to be in alphabetical order. Try typing the following into the interactive shell:

```
spam = [5, 3, 4, 1, 2]
spam.sort()
```

```
spam
```

```
>>> spam = [5, 3, 4, 1, 2]
>>> spam.sort()
>>> spam
[1, 2, 3, 4, 5]
>>>
```

Notice that the `sort()` method does not *return* a sorted list, but rather just sorts the list it is called on. You would never want to use this line of code: `return spam.sort()` because that would return the value `None` (which is what `sort()` returns). Instead you would want a separate line `spam.sort()` and then the line `return spam`.

The reason we want to sort the `clue` list is because we might return extra clues that we did not intend based on the order of the clues. If `clue` had the value `['Pico', 'Fermi', 'Pico']`, that would tell us that the center digit of our guess is in the correct position. Since the other two clues are both Pico, then we know that all we have to do is swap the first and third digit and we have the secret number. But if the clues are always sorted in alphabetical order, the player would not know which number the Fermi clue refers to.

The `join()` String Method

```
27.     return ' '.join(clue)
```

The `join()` string method returns a string of each item in the list argument joined together. The string that the method is called on (on line 27, this is a single space, `' '`) appears in between each item in the list. So the string that is returned on line 27 is each string in `clue` combined together with a single space in between each string.

For an example, type the following into the interactive shell:

```
'x'.join(['hello', 'world'])

'ABCDEF'.join(['x', 'y', 'z'])

''.join(['My', 'name', 'is', 'Sam'])
```

```
>>> 'x'.join(['hello', 'world'])
'helloxworld'
>>> 'ABCDEF'.join(['x', 'y', 'z'])
'xABCDEFyABCDEFz'
>>> ''.join(['My', 'name', 'is', 'Sam'])
'MynameisSam'
>>>
```

Code Explanation Continued...

```
29. def isOnlyDigits(num):
30.     # Returns True if num is a string made up only of
    digits. Otherwise returns False.
31.     if num == '':
32.         return False
```

The `isOnlyDigits()` is a small function that will help us determine if the player entered a guess that was only made up of numbers. To do this, we will check each individual letter in the string named `num` and make sure it is a number.

Line 31 does a quick check to see if we were sent the blank string, and if so, we return `False`.

```
34.     for i in num:
35.         if i not in '0 1 2 3 4 5 6 7 8 9'.split():
36.             return False
37.
38.     return True
```

We use a `for` loop on the string `num`. The value of `i` will have a single character from the `num` string on each iteration. Inside the for-block, we check if `i` does not exist in the list returned by `'0 1 2 3 4 5 6 7 8 9'.split()`. If it doesn't, we know that there is a character in `num` that is something besides a number. In that case, we should return the value `False`.

If execution continues past the `for` loop, then we know that every character in `num` is a number. So we return the value `True`.

```
40. def playAgain():
```

```
41.     # This function returns True if the player wants to
    play again, otherwise it returns False.
42.     print 'Do you want to play again? (yes or no)'
43.     return raw_input().lower().startswith('y')
```

The `playAgain()` function is the same one we used in Hangman and Tic Tac Toe. The long expression on line 43 will evaluate to either `True` or `False`. The return value from the call to the `raw_input()` function is a string that has its `lower()` method called on it. The `lower()` method returns another string (the lowercase string) and that string has its `startswith()` method called on it, passing the argument `'y'`.

```
45. NUMDIGITS = 3
46. MAXGUESS = 10
47.
48. print 'I am thinking of a %s-digit number. Try to guess
    what it is.' % (NUMDIGITS)
49. print 'Here are some clues:'
50. print 'When I say:    That means:'
51. print '  Pico         One digit is correct but in the
    wrong position.'
52. print '  Fermi        One digit is correct and in the
    right position.'
53. print '  Bagels       No digit is correct.'
```

This is the actual start of the program. Instead of hard-coding three digits as the size of the secret number, we will use the constant variable `NUMDIGITS`. And instead of hard-coding a maximum of ten guesses that the player can make, we will use the constant variable `MAXGUESS`. (This is because if we increase the number of digits the secret number has, we also might want to give the player more guesses. We put the variable names in all capitals to show they are constant by convention.)

The print statements will tell the player the rules of the game and what the Pico, Fermi, and Bagels clues mean. Line 48's `print` statement has the code `% (NUMDIGITS)` added to the end and `%s` inside the string. This is a technique know as string interpolation.

String Interpolation

String interpolation is another shortcut, like augmented assignment operators. Normally, if you want to use the string values inside variables in another string, you have to use the + concatenation operator:

```
name = 'Alice'
```

```
event = 'party'
where = 'the pool'
when = 'Saturday'
time = '6:00pm'
print 'Hello, ' + name + '. Will you go to the ' + event + '
at ' + where + ' this ' + when + ' at ' + time + '?'
```

```
>>> name = 'Alice'
>>> event = 'party'
>>> where = 'the pool'
>>> when = 'Saturday'
>>> time = '6:00pm'
>>> print 'Hello, ' + name + '. Will you go to the ' + event + ' at ' + where + '
Hello, Alice. Will you go to the party at the pool this Saturday at 6:00pm?
>>> 
```

As you can see, it can be very hard to type a line that concatenates several strings together. Instead, you can use **string interpolation**, which lets you put placeholders like `%s` (these placeholders are called **conversion specifiers**), and then put all the variable names at the end. Each `%s` is replaced with the value in the variable at the end of the line. For example, the following code does the same thing as the above code:

```
name = 'Alice'
event = 'party'
where = 'the pool'
when = 'Saturday'
time = '6:00pm'
print 'Hello, %s. Will you go to the %s at %s this %s at %
s?' % (name, event, where, when, time)
```

String interpolation can make your code much easier to type and read, rather than using several + concatenation operators.

The final line has the `print` keyword, followed by the string with conversion specifiers, followed by the % sign, followed by a set of parentheses with the variables in them. The first variable `name` will be used for the first `%s`, the second variable with the second `%s` and so on. The Python interpreter will give you an error if you do not have the same number of `%s` conversion specifiers as you have variables.

Another benefit of using string interpolation instead of string concatenation is that interpolation works with any data type, not just strings. All values are automatically converted to the string data type. (This is what the `s` in `%s` stands for.) If you typed this code into the shell, you'd get an error:

```
spam = 42
print 'Spam == ' + spam
```

```
>>> spam = 42
>>> print 'Spam == ' + spam

Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    print 'Spam == ' + spam
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

You get this error because string concatenation can only combine two strings, and `spam` is an integer. You would have to remember to put `str(spam)` in there instead. But with string interpolation, you can have any data type. Try typing this into the shell:

```
spam = 42
print 'Spam == %s' % (spam)
```

```
>>> spam = 42
>>> print 'Spam == %s' % (spam)
Spam == 42
>>>
```

As you can see, using string interpolation instead of string concatenation is much easier because you don't have to worry about the data type of the variable. Also, string interpolation can be done on any strings, not just strings used in `print` statements.

String interpolation is also known as **string formatting**.

Code Explanation Continued...

```
while True:
    secretNum = getSecretNum(NUMDIGITS)
    print 'I have thought up a number. You have %s guesses to get it.' % (MAXGUESS)

    numGuesses = 1
    while numGuesses <= MAXGUESS:
```

We start with a `while` loop that has a condition of `True`, meaning it will loop forever until we execute a `break` statement. Inside the loop, we get a secret number from our `getSecretNum()` function (passing it `NUMDIGITS` to tell how many digits we want the secret number to have) and assign it to `secretNum`. Remember that `secretNum` is a string, not an integer.

We tell the player how many digits is in our secret number by using string interpolation instead of string concatenation. We set a variable `numGuesses` to `1`, to denote that this is the first guess. Then we enter a new `while` loop which will keep looping as long as `numGuesses` is less than or equal to `MAXGUESS`.

Notice that this second `while` loop on line 60 is inside another `while` loop that started on line 55. Whenever we have these loops-inside-loops, we call them **nested loops**. You should know that any `break` or `continue` statements will only break or continue out of the innermost loop, and not any of the outer loops.

```
61.         guess = ''
62.         while len(guess) != NUMDIGITS or not
    isOnlyDigits(guess):
63.             print 'Guess #%s: ' % (numGuesses)
64.             guess = raw_input()
```

The `guess` variable will hold the player's guess. We will keep looping and asking the player for a guess until the player enters a guess that 1) has the same number of digits as the secret number and 2) is made up only of digits. This is what the `while` loop on line 62 is for. We set `guess` as the blank string on line 61 so that the `while` loop's condition is `False` the first time, ensuring that we enter the loop at least once.

```
66.         clue = getClues(guess, secretNum)
67.         print clue
68.         numGuesses += 1
```

After execution gets past the `while` loop on line 62, we know that `guess` contains a valid guess. We pass this and the secret number in `secretNum` to our `getClues()` function. It returns a string that contains our clues, which we will display to the player. We then increment `numGuesses` by 1 using the augmented assignment operator for addition.

```
70.         if guess == secretNum:
71.             break
72.         if numGuesses > MAXGUESS:
73.             print 'You ran out of guesses. The answer
    was %s.' % (secretNum)
```

If `guess` is the same value as `secretNum`, then we know the player has correctly guessed the secret number and we can break out of this loop (the `while` loop that was started on line 60). If not, then execution continues to line 72, where we check to see if the player ran out of guesses. If so, then we tell the player that they have lost and what the secret number was. We know that the condition for the `while` loop on line 55 will be `False`, so there is no need for a `break` statement.

At this point, execution jumps back to the `while` loop on line 60 where we let the player have another guess. If the player ran out of guesses (or we broke out of the loop with the `break` statement on line 71), then execution would proceed to line 75.

```
75.     if not playAgain():
76.         break
77.
```

After leaving the `while` loop on line 60, we ask the player if want to play again by calling our `playAgain()` function. If `playAgain()` returns `False`, then we should break out of the `while` loop that was started on line 55. Since there is no more code after this loop, the program terminates.

If `playAgain()` returned `True`, then we would not execute the `break` statement and execution would jump back to line 55. A new secret number would be generated so that the player can play a new game.

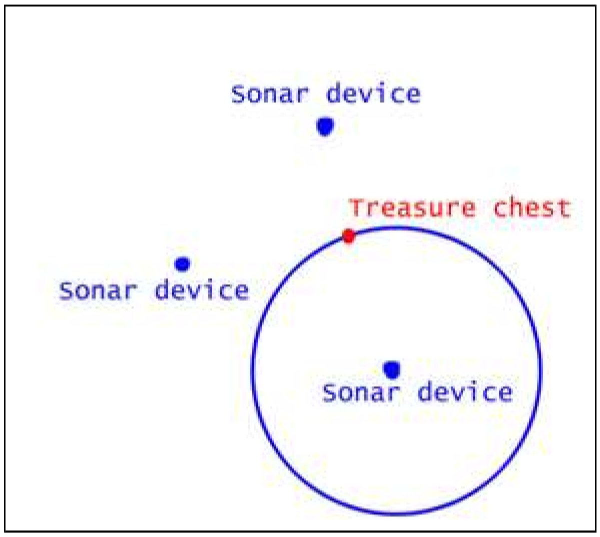
Things Covered In This Chapter:

- Hard-coding
- Augmented Assignment Operators, +=, -=, *=, /=
- The sort() List Method
- The join() List Method
- String Interpolation (also called String Formatting)
- Conversion Specifier %s
- Nested Loops

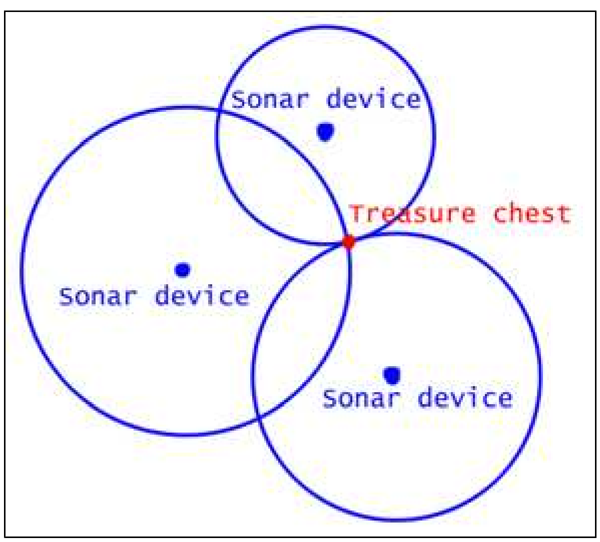
Chapter 8 - Sonar

Sonar is a technology that ships use to locate objects under the sea. In this chapter's game, the player places sonar devices at various places in the ocean to locate sunken treasure chests. The sonar devices in our game can tell the player how far away a treasure chest is from the sonar device, but not in what direction. But by placing multiple sonar devices down, the player can figure out where exactly the treasure chest is. There are three chests to collect, but the player has only sixteen sonar devices to use to find them.

Imagine that we could not see the treasure chest (the red dot) in the following picture. Because each sonar device can only find the distance but not direction, the possible places the treasure could be is anywhere in a ring around the sonar device:

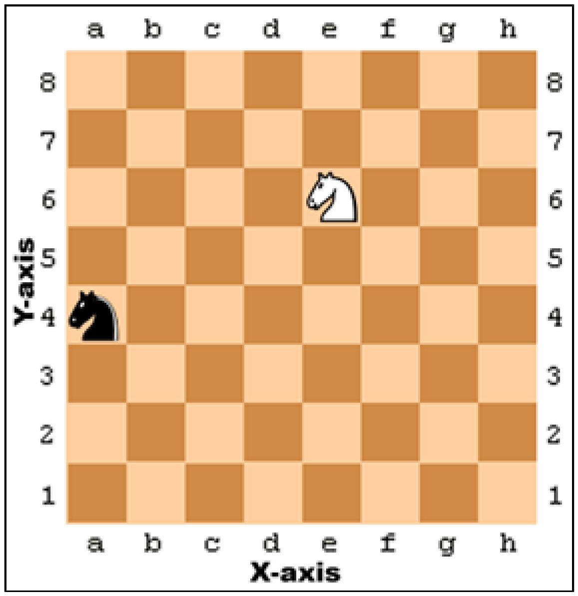


But if we have multiple sonar devices working together, we can narrow it down to an exact place where all the rings intersect each other:



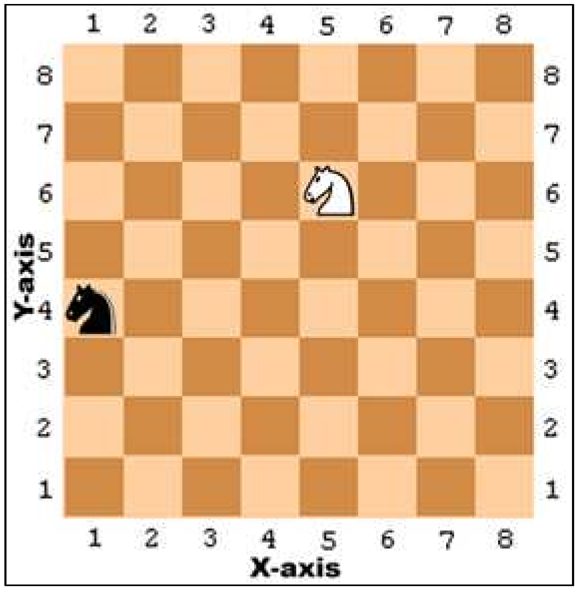
Grids and Cartesian Coordinates

A problem in many games is how to talk about exact points on the board. A common way of solving this is by marking each individual row and column on a board with a letter and a number. Here is a chess board that has each row and each column marked.



In chess, the knight piece looks like a horse. The white knight is located at the point e, 6 and the black knight is located at point a, 4. We can also say that every space on row 7 or every space in column c is empty.

A grid with labeled rows and columns like the chess board is called a **Cartesian coordinate system**. By using a row label and column label, we can give a coordinate that is to exactly one and only one space. This can really help us describe to a computer the exact location we want. If you have learned about Cartesian coordinate systems in math class, you may know that usually we have numbers for both the rows and columns. This is handy, because otherwise after the 26th column we would run out of letters. That board would look like this:



The numbers going left and right that describe the columns are part of the **X-axis**. The numbers going up and down that describe the rows are part of the **Y-axis**. When we describe coordinates, we always say the X coordinate first, followed by the Y coordinate. That means the white knight in the above picture is located at the coordinate 5, 6. The black knight is located at the coordinate 1, 4.

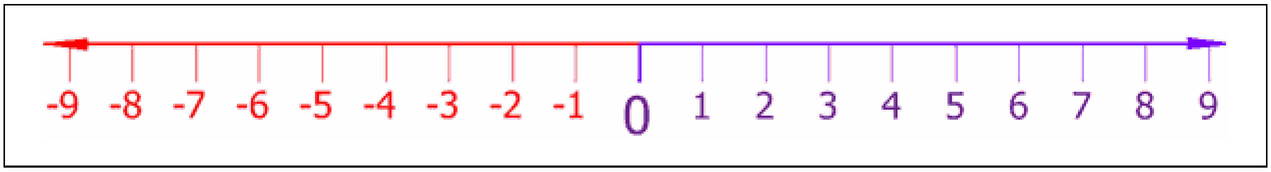
Notice that for the black knight to move to the white knight's position, the black knight must move up two spaces, and then to the right by four spaces. (Or move right four spaces and then move up two spaces.) But we don't need to look at the board to figure this out. If we know the white knight is located at 5, 6 and the black knight is located at 1, 4, then we can just use subtraction to figure out this information.

Subtract the black knight's X coordinate and white knight's X coordinate: 5 - 1 = 4. That means the black knight has to move along the X-axis by four spaces.

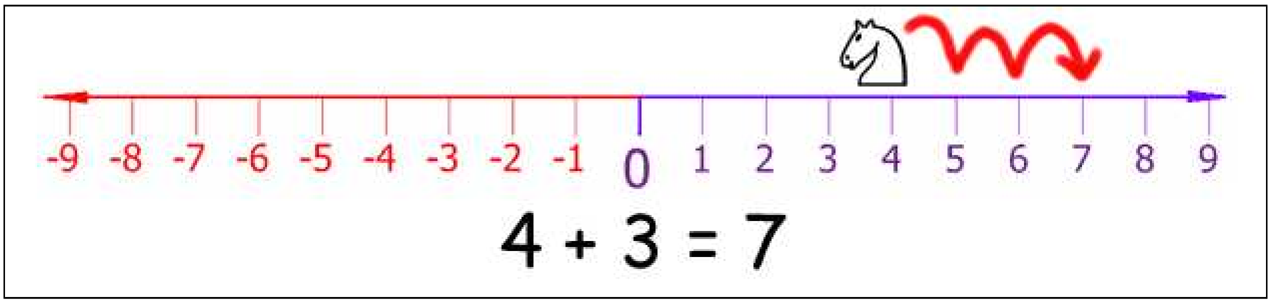
Subtract the black knight's Y coordinate and white knight's Y coordinate: 6 - 4 = 2. That means the black knight has to move along the Y-axis by two spaces.

Negative Numbers

Another concept that Cartesian coordinates use is negative numbers. **Negative numbers** are numbers that are smaller than zero. We put a minus sign in front of a number to show that it is a negative number. -1 is smaller than 0. -2 is smaller than -1. -3 is smaller than -2. If you think of regular numbers (called **positive numbers**) as starting from 1 and increasing, you can think of negative numbers as starting from -1 and decreasing. 0 itself is not positive or negative. In this picture, you can see the positive numbers increasing to the right and the negative numbers decreasing to the left:

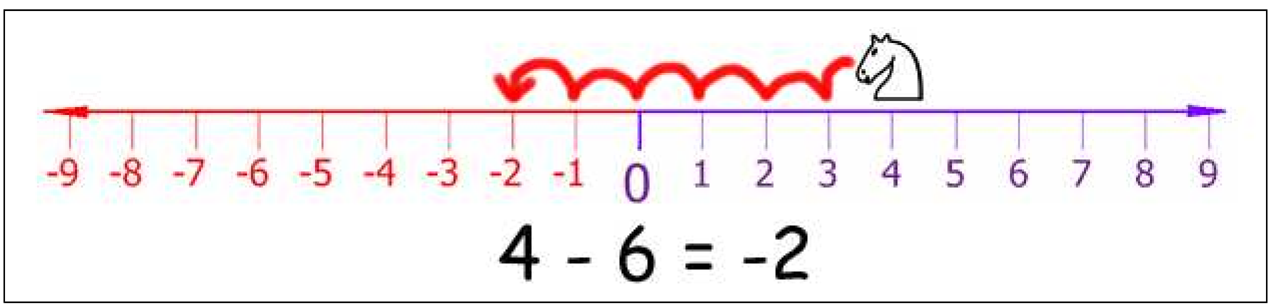


The number line is really useful for doing subtraction and addition with negative numbers. The expression 4 + 3 can be thought of as the white knight starting at position 4 and moving 3 spaces over to the right (addition means increasing, which is in the right direction).



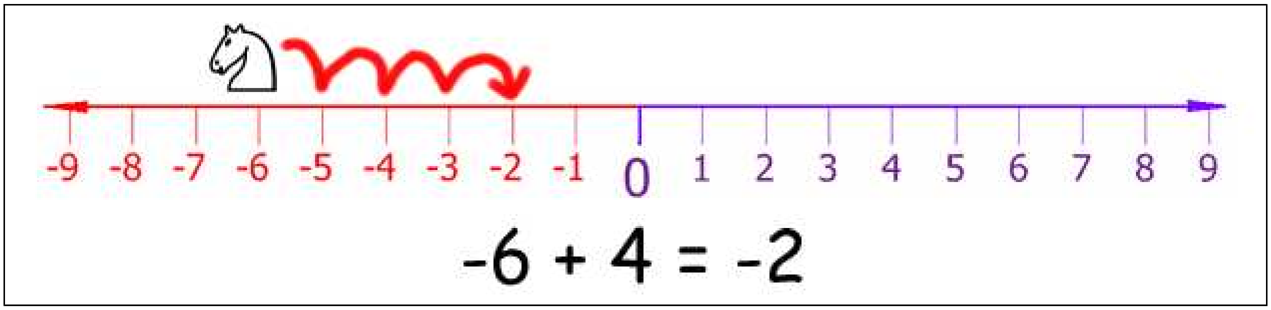
As you can see, the white knight ends up at position 7. This makes sense, because 4 + 3 is 7.

Subtraction can be done by moving the white knight to the left. Subtraction means decreasing, which is in the left direction. 4 - 6 would be the white knight starting at position 4 and moving 6 spaces to the left:

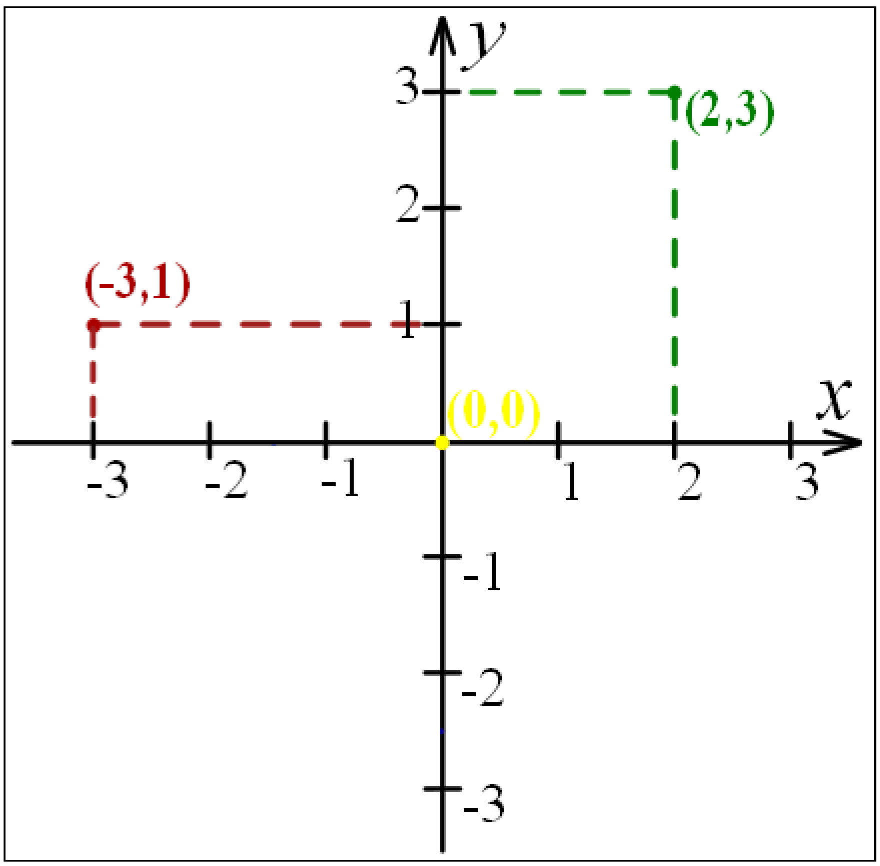


The white knight ends up at position -2. That means 4 - 6 equals -2.

If we add or subtract a negative number, the white knight would move in the *opposite* direction. If you add a negative number, the knight moves to the *left*. If you subtract a negative number, the knight moves to the *right*. The expression -6 - -4 would be equal to -2. The knight starts at -6 and moves to the *right* by 4 spaces. Notice that -6 - -4 has the same answer as -6 + 4.



The number line is the same as the X-axis. If we made the number line go up and down instead of left and right, it would model the Y-axis. Adding a positive number (or subtracting a negative number) would move the knight up the number line, and subtracting a positive number (or adding a negative number) would move the knight down. When we put these two number lines together, we have a Cartesian coordinate system.

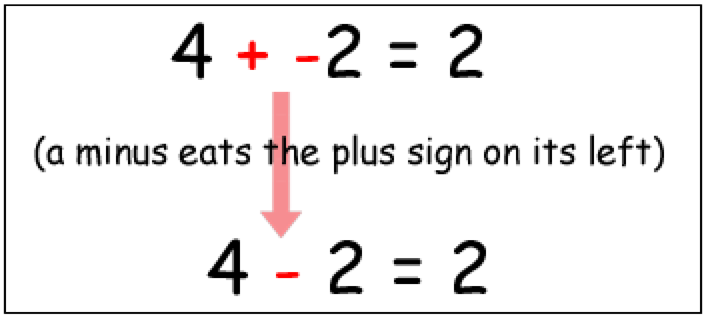


The 0, 0 coordinate has a special name, the **origin**.

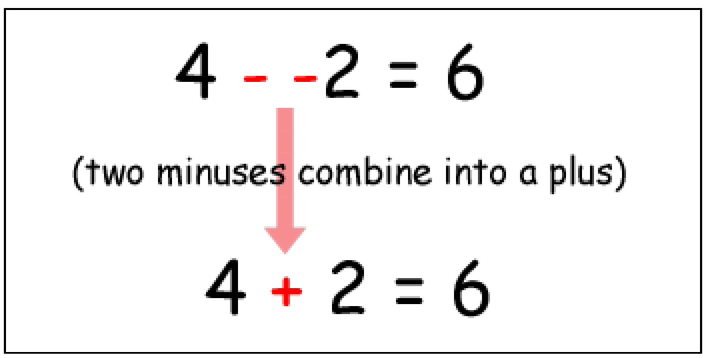
Changing the Signs

Subtracting negative numbers or adding negative numbers seems easy when you have a number line in front of you, but it can be easy when you only have the numbers too. Here are three tricks you can do to make evaluating these expressions easier to do.

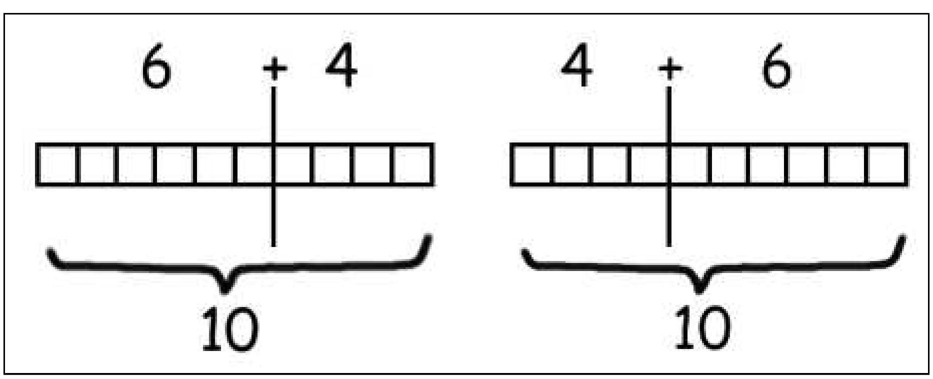
The first is if you are adding a negative number, for example; 4 + -2. The first trick is "a minus eats the plus sign on its left". When you see a minus sign with a plus sign on the left, you can replace the plus sign with a minus sign. The answer is still the same, because adding a negative value is the same as subtracting a positive value. 4 + -2 and 4 - 2 both evaluate to 2.



The second trick is if you are subtracting a negative number, for example, 4 - -2. The second trick is "two minuses combine into a plus". When you see the two minus signs next to each other without a number in between them, they can combine into a plus sign. The answer is still the same, because subtracting a negative value is the same as adding a positive value.

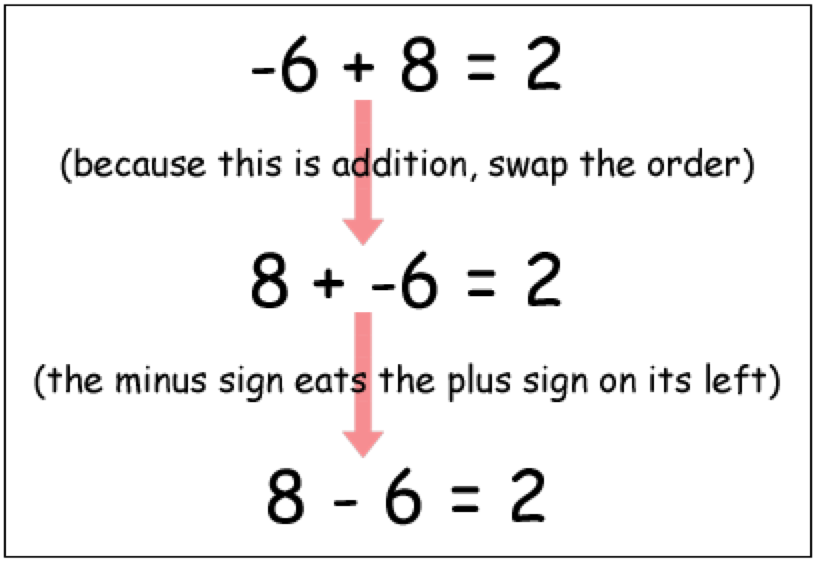


A third trick is to remember that when you add two numbers like 6 and 4, it doesn't matter what order they are in. (This is called the **commutative property** of addition.) That means that 6 + 4 and 4 + 6 both equal the same value, 10.



Say you are adding a negative number and a positive number, like -6 + 8. Because you are adding numbers, you can swap the order of the numbers without changing the answer. -6 + 8 is the same as 8 +

-6. But when you look at 8 + -6, you see that the minus sign can eat the plus sign to its left, and the problem becomes 8 - 6 = 2. But this means that -6 + 8 is also 2! We've rearranged the problem to have the same answer, but made it easier to solve.



Of course, you can always use the interactive shell as a calculator to evaluate these expressions. It is still very useful to know the above three tricks when adding or subtracting negative numbers. After all, you won't always be in front of a computer with Python all the time!

```
>>> 4 + -2
2
>>> -4 + 2
-2
>>> -4 + -2
-6
>>> 4 - -2
6
>>> -4 - 2
-6
>>> -4 - -2
-2
>>>
```

Absolute Values

The **absolute value** of a number is the number without the negative sign in front of it. This means that positive numbers do not change, but negative numbers become positive. For example, the absolute value of -4 is 4. The absolute value of -7 is 7. The absolute value of 5 (which is positive) is 5.

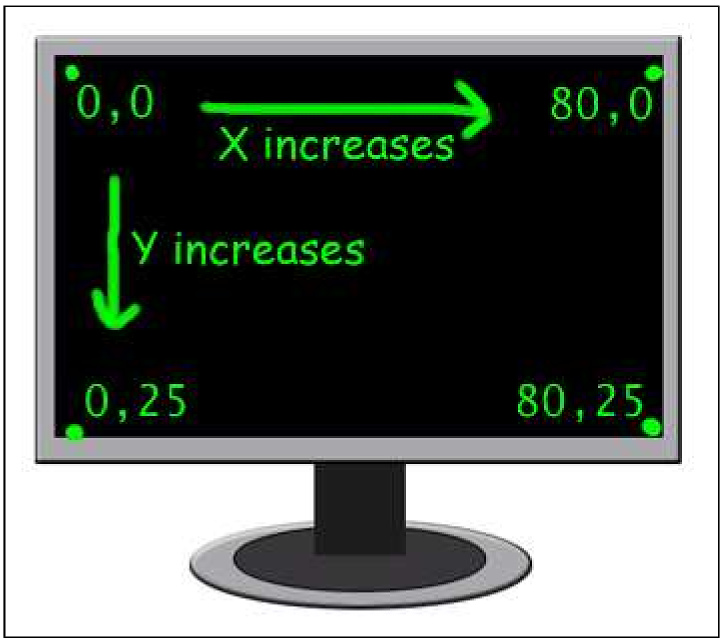
We can find how far away two things on a number line are from each other by taking the absolute value of their difference. Imagine that the white knight is at position 4 and the black knight is at position -2. To find out the distance between them, you would find the difference by subtracting their positions

and taking the absolute value of that number.

It works no matter what the order of the numbers is. -2 - 4 (that is, negative two minus four) is -6, and the absolute value of -6 is 6. However, 4 - -2 (that is, four minus negative two) is 6, and the absolute value of 6 is 6. Using the absolute value of the difference is a good way of finding the distance between two points on a number line (or axis).

Coordinate System of a Computer Monitor

It is common that computer monitors use a coordinate system that has the origin (0, 0) at the top left corner of the screen, which increases going down and to the right. There are no negative coordinates. This is because text is printed starting at the top left, and is printed going to the right and downwards. Most computer graphics use this coordinate system, and we will use it in our games. Also it is common to assume that monitors can display 80 text characters per row and 25 text characters per column. This used to be the maximum screen size that monitors could support. While today's monitors can usually display much more text, we will not assume that the user's screen is bigger than 80 by 25.



Sample Run

```
S O N A R !

Would you like to view the instructions? (yes/no)
no
             1         2         3         4         5
   012345678901234567890123456789012345678901234567890123456789

 0 `~~~`~~~`~`~~`~~~~~`~``~~~~`~`~~~`~``~``~~```~`~```~`~~~~`` 0
 1 ~`~~~```~~~~`~`~~`~``~`~~```~`~`~~`~`~~~~~~`~`````~`~~`~~~~` 1
 2 `~``~``~~~`~``~`~`~``~````~~~~~~~~~`~`~~`~``~~~~~```~~`~``` 2
 3 ``~`~~``~`~``~`~`~`~~`~`~~`~`~``~~~`~``~````~``````~~~~``~`` 3
 4 ``~~`~~~``~``~~```~`~`~`~``~~~``~~```~`~~`~~`~`~`~~`~~~~``` 4
 5 ~~```~~~`~`~~``~`~``~```~`~~`~~~~~`~~``~`~`~~~`~~`~`~`~`~~~` 5
 6 ``~~`````~~~~`~`~~~```~~~~`~~`~~`~~```~~`~~~`~~~``~`~~~``~~~ 6
```

```
 7 `~`````````~```~``~``~~`~~~~`~~``~``~~~```~`~~`~``~``~~```~~ 7
 8 `~````~```~`~~`~~~`~~``~~~``~`~``~~~``~`~`````~`~~```~`~~~~` 8
 9 ~```~~`~`~``~``~~``~``~```~`~``~~~~`~`~`~~~`~`~`~`~~~``~~``` 9
10 ```~`~```~~`~``~`~~`~``~````~``~~~`~~`~~``~~~~`~~~`~`~~````~ 10
11 ```~```~~~`~```~~`~~~`~`````~`~~`~`~~`~~`~`~~`~~~````~````~` 11
12 ~~~`~`~~~``~~~~~~`~~~``~`~`~~`~`~~`~```~~~```~~`~~`~``~``~`~ 12
13 `~~````~~``~```~~~`~```~`~~~~~~~~~`~~``~~~~~`````~`~`~``~~~~ 13
14 `~~`~`~````~```~`~`~```~~`~~~~`~```~``~``~``~~~````~~``````~ 14

   012345678901234567890123456789012345678901234567890123456789
             1         2         3         4         5
You have 16 sonar devices left. 3 treasure chests remaining.
Where do you want to drop the next sonar device? (0-59 0-14) (or type quit)
```
10 10
```
             1         2         3         4         5
   012345678901234567890123456789012345678901234567890123456789

 0 `~~~`~~~`~`~~`~~~~~`~``~~~~`~`~~~`~``~``~~```~`~```~`~~~~`` 0
 1 ~`~~~```~~~~`~`~~`~``~`~~```~`~`~~`~`~~~~~~`~````~`~~`~~~~` 1
 2 `~``~``~~~`~``~`~`~``~````~~~~~~~~~`~`~~`~``~~~~~```~~`~``` 2
 3 ``~`~~``~`~``~`~`~`~~`~`~~`~`~``~~~`~``~```~`````~~~~``~`` 3
 4 ``~~`~~~``~``~~```~`~`~`~``~~~``~~```~`~~`~~`~`~`~~`~~~~``` 4
 5 ~~```~~~`~`~~``~`~``~```~`~~`~~~~~`~~``~`~`~~~`~~`~`~`~`~~~` 5
 6 ``~~````~~~~`~`~~~``~~~~`~~`~~`~~``~~`~~~`~~~``~`~~~``~~~ 6
 7 `~`````````~```~``~``~~`~~~~`~~``~``~~~```~`~~`~``~``~~```~~ 7
 8 `~````~```~`~~`~~~`~~``~~~``~`~``~~~``~`~`````~`~~```~`~~~~` 8
 9 ~```~~`~`~``~``~~``~``~```~`~``~~~~`~`~`~~~`~`~`~`~~~``~~``` 9
10 ```~`~```~5`~``~`~~`~``~````~``~~~`~~`~~``~~~~`~~~`~`~~````~ 10
11 ```~```~~~`~```~~`~~~`~`````~`~~`~`~~`~~`~`~~`~~~````~````~` 11
12 ~~~`~`~~~``~~~~~~`~~~``~`~`~~`~`~~`~```~~~```~~`~~`~``~``~`~ 12
13 `~~````~~``~```~~~`~```~`~~~~~~~~~`~~``~~~~~`````~`~`~``~~~~ 13
14 `~~`~`~````~```~`~`~```~~`~~~~`~```~``~``~``~~~````~~``````~ 14

   012345678901234567890123456789012345678901234567890123456789
             1         2         3         4         5
Treasure detected at a distance of 5 from the sonar device.
You have 15 sonar devices left. 3 treasure chests remaining.
Where do you want to drop the next sonar device? (0-59 0-14) (or type quit)
```
15 6
```
             1         2         3         4         5
   012345678901234567890123456789012345678901234567890123456789

 0 `~~~`~~~`~`~~`~~~~~`~``~~~~`~`~~~`~``~``~~```~`~```~`~~~~`` 0
 1 ~`~~~```~~~~`~`~~`~``~`~~```~`~`~~`~`~~~~~~`~````~`~~`~~~~` 1
 2 `~``~``~~~`~``~`~`~``~````~~~~~~~~~`~`~~`~``~~~~~```~~`~``` 2
 3 ``~`~~``~`~``~`~`~`~~`~`~~`~`~``~~~`~``~```~`````~~~~``~`` 3
 4 ``~~`~~~``~``~~```~`~`~`~``~~~``~~```~`~~`~~`~`~`~~`~~~~``` 4
 5 ~~```~~~`~`~~``~`~``~```~`~~`~~~~~`~~``~`~`~~~`~~`~`~`~`~~~` 5
 6 ``~~````~~~~`~4~~~```~~~~`~~`~~`~~``~~`~~~`~~~``~`~~~``~~~ 6
 7 `~`````````~```~``~``~~`~~~~`~~``~``~~~```~`~~`~``~``~~```~~ 7
 8 `~````~```~`~~`~~~`~~``~~~``~`~``~~~``~`~`````~`~~```~`~~~~` 8
 9 ~```~~`~`~``~``~~``~``~```~`~``~~~~`~`~`~~~`~`~`~`~~~``~~``` 9
10 ```~`~```~5`~``~`~~`~``~````~``~~~`~~`~~``~~~~`~~~`~`~~````~ 10
11 ```~```~~~`~```~~`~~~`~`````~`~~`~`~~`~~`~`~~`~~~````~````~` 11
12 ~~~`~`~~~``~~~~~~`~~~``~`~`~~`~`~~`~```~~~```~~`~~`~``~``~`~ 12
13 `~~````~~``~```~~~`~```~`~~~~~~~~~`~~``~~~~~`````~`~`~``~~~~ 13
14 `~~`~`~````~```~`~`~```~~`~~~~`~```~``~``~``~~~````~~``````~ 14

   012345678901234567890123456789012345678901234567890123456789
             1         2         3         4         5
Treasure detected at a distance of 4 from the sonar device.
You have 14 sonar devices left. 3 treasure chests remaining.
Where do you want to drop the next sonar device? (0-59 0-14) (or type quit)
```
15 10
```
             1         2         3         4         5
   012345678901234567890123456789012345678901234567890123456789

 0 `~~~`~~~`~`~~`~~~~~`~``~~~~`~`~~~`~``~``~~```~`~```~`~~~~`` 0
 1 ~`~~~```~~~~`~`~~`~``~`~~```~`~`~~`~`~~~~~~`~````~`~~`~~~~` 1
 2 `~``~``~~~`~``~`~`~``~````~~~~~~~~~`~`~~`~``~~~~~```~~`~``` 2
 3 ``~`~~``~`~``~`~`~`~~`~`~~`~`~``~~~`~``~```~`````~~~~``~`` 3
 4 ``~~`~~~``~``~~```~`~`~`~``~~~``~~```~`~~`~~`~`~`~~`~~~~``` 4
 5 ~~```~~~`~`~~``~`~``~```~`~~`~~~~~`~~``~`~`~~~`~~`~`~`~`~~~` 5
 6 ``~~````~~~~`~O~~~```~~~~`~~`~~`~~``~~`~~~`~~~``~`~~~``~~~ 6
 7 `~`````````~```~``~``~~`~~~~`~~``~``~~~```~`~~`~``~``~~```~~ 7
```

```
 8 `~````~```~`~~`~~~`~~``~~~``~`~``~~~``~`~`````~`~~```~`~~~~` 8
 9 ~```~~`~`~``~``~~``~``~```~`~``~~~~`~`~`~~~`~`~`~`~~~``~~``` 9
10 ```~`~```~O`~``O`~~`~``~````~``~~~`~~`~~``~~~~`~~~`~`~~````~ 10
11 ```~```~~~`~```~~`~~~`~`````~`~~`~`~~`~~`~`~~`~~~````~````~` 11
12 ~~~`~`~~~``~~~~~~`~~~``~`~`~~`~`~~`~```~~~```~~`~~`~``~``~`~ 12
13 `~~````~~``~```~~~`~```~`~~~~~~~~~`~~``~~~~~`````~`~`~``~~~~ 13
14 `~~`~`~````~```~`~`~```~~`~~~~`~```~``~``~``~~~````~~``````~ 14

   012345678901234567890123456789012345678901234567890123456789
             1         2         3         4         5
You have found a sunken treasure chest!
You have 13 sonar devices left. 2 treasure chests remaining.
Where do you want to drop the next sonar device? (0-59 0-14) (or type quit)


...skipped over for brevity....


             1         2         3         4         5
   012345678901234567890123456789012345678901234567890123456789

 0 `~~~`~~~`~`~~`~~~~~`~``~~~~`~`~~~`~``~``~~````~`~```~`~~~~`` 0
 1 ~`~~~```~~~~`~`~~`~``~`~~```~O~`~~`~`~~~~~~`~`````~`~~`~~~~` 1
 2 `~``~``~~~`~``~`~`~``~`````~~O~~~O~~`~`~~`~``~~~~~```~~`~``` 2
 3 ``~3~~``8`~``~`~`~`~~`~`~~`~`~``~~~`~`O~````~``````~~~~``~`` 3
 4 ``~~`~~~``~``~~````~`~`~`~O`~~O``~~```~`~~`~~`~`~`~~`~~~~``` 4
 5 ~~```~~~`~`~~``~`~``~```~`~~`~~~~~`~~``~`~`~~~`~~`~`~`~`~~~` 5
 6 ``~~`````~~~~`~O~~~```~~~~`~~`~~`~~```~~`~~~`~~~``O`~~~``~~~ 6
 7 `~`````````~```~``~``~~`~~~~`~~``~``~~~```~`~~`~``~``~~```~~ 7
 8 `~````~```~`~~`~~~`~~``~~~``~`~``~~~``~`O```0`~`~~```~`~~~~` 8
 9 ~```~~`~`~``~``~~``~``~```~O~``~~~~`~`~`~~~`~`~`~`~~~``~~``` 9
10 ```~`~```~O`~``O`~~`~``~````~``~~~`~~`~~``~~~~`~~~`~`~~````~ 10
11 ```~```~~~`~```~~`~~~`~`````~`~~`~`~~`~~`~`~~`~~~````~````~` 11
12 ~~~`~`~~~``~~~~~~`~~~``~`~`~~`~`~~`~```~~~```~~`~~`~``~``~`~ 12
13 `~~````~~``~```~~~`~```~`~~~~~~~~~`~~``~~~~~`````~`~`~``~~~~ 13
14 `~~`~`~````~```~`~`~```~~`~~~~`~```~``~``~``~~~````~~``````~ 14

   012345678901234567890123456789012345678901234567890123456789
             1         2         3         4         5
Treasure detected at a distance of 4 from the sonar device.
We've run out of sonar devices! Now we have to turn the ship around and head
for home with treasure chests still out there! Game over.
    The remaining chests were here:
    0, 4
Do you want to play again? (yes or no)
no
```

Source Code

Knowing about Cartesian coordinates, number lines, negative numbers, and absolute values will help us out with our Sonar game. Here is the source code for the game. Type it into a new file, then save the file as sonar.py and run it by pressing the F5 key. You do not need to understand the code to type it in or play the game, the source code will be explained later.

sonar.py

```
# Sonar

import random
import sys

def drawBoard(board):
```

```
    # Draw the board data structure.

    hline = '    ' # initial space for the numbers down the left side of the board
    for i in range(1, 6):
        hline += (' ' * 9) + str(i)

    # print the numbers across the top
    print hline
    print '   ' + ('0123456789' * 6)
    print

    # print each of the 15 rows
    for i in range(15):
        # single-digit numbers need to be padded with an extra space
        if i < 10:
            extraSpace = ' '
        else:
            extraSpace = ''
        print '%s%s %s %s' % (extraSpace, i, getRow(board, i), i)

    # print the numbers across the bottom
    print
    print '   ' + ('0123456789' * 6)
    print hline


def getRow(board, row):
    # Return a string from the board data structure at a certain row.
    boardRow = ''
    for i in range(60):
        boardRow += board[i][row]
    return boardRow

def getNewBoard():
    # Create a new 60x15 board data structure.
    board = []
    for x in range(60): # the main list is a list of 60 lists
        board.append([])
        for y in range(15): # each list in the main list has 15 single-character strings
            # use different characters for the ocean to make it more readable.
```

```
            if random.randint(0, 1) == 0:
                board[x].append('~')
            else:
                board[x].append('`')
    return board

def getRandomChests(numChests):
    # Create a list of chest data structures (two-item lists of x, y int coordinates)
    chests = []
    for i in range(numChests):
        chests.append([random.randint(0, 59), random.randint(0, 14)])
    return chests

def isValidMove(x, y):
    # Return True if the coordinates are on the board, otherwise False.
    return x >= 0 and x <= 59 and y >= 0 and y <= 14

def makeMove(board, chests, x, y):
    # Change the board data structure with a sonar device character. Remove treasure chests
    # from the chests list as they are found. Return False if this is an invalid move.
    # Otherwise, return the string of the result of this move.
    if not isValidMove(x, y):
        return False

    smallestDistance = 100 # any chest will be closer than 100.
    for cx, cy in chests:
        if abs(cx - x) > abs(cy - y):
            distance = abs(cx - x)
        else:
            distance = abs(cy - y)

        if distance < smallestDistance: # we want the closest treasure chest.
            smallestDistance = distance

    if smallestDistance == 0:
        # xy is directly on a treasure chest!
        chests.remove([x, y])
        return 'You have found a sunken treasure chest!'
```

```
    else:
        if smallestDistance < 10:
            board[x][y] = str(smallestDistance)
            return 'Treasure detected at a distance of %s from the sonar device.' % (smallestDistance)
        else:
            board[x][y] = 'O'
            return 'Sonar did not detect anything. All treasure chests out of range.'


def enterPlayerMove():
    # Let the player type in her move. Return a two-item list of int xy coordinates.
    print 'Where do you want to drop the next sonar device? (0-59 0-14) (or type quit)'
    while True:
        move = raw_input()
        if move.lower() == 'quit':
            print 'Thanks for playing!'
            sys.exit()

        move = move.split()
        if len(move) == 2 and move[0].isdigit() and move[1].isdigit() and isValidMove(int(move[0]), int(move[1])):
            return [int(move[0]), int(move[1])]
        print 'Enter a number from 0 to 59, a space, then a number from 0 to 14.'


def playAgain():
    # This function returns True if the player wants to play again, otherwise it returns False.
    print 'Do you want to play again? (yes or no)'
    return raw_input().lower().startswith('y')


def showInstructions():
    print '''Instructions:
You are the captain of the Simon, a treasure-hunting ship. Your current mission
is to find the three sunken treasure chests that are lurking in the part of the
ocean you are in and collect them.

To play, enter the coordinates of the point in the
```

```
ocean you wish to drop a sonar device. The sonar can find out how far away the closest chest is to it.
For example, the d below marks where the device was dropped, and the 2's
represent distances of 2 away from the device. The 4's represent
distances of 4 away from the device.

    444444444
    4       4
    4 22222 4
    4 2   2 4
    4 2 d 2 4
    4 2   2 4
    4 22222 4
    4       4
    444444444
Press enter to continue...'''
    raw_input()

    print '''For example, here is a treasure chest (the c) located a distance of 2 away
from the sonar device (the d):

    22222
    c   2
    2 d 2
    2   2
    22222

The point where the device was dropped will be marked with a 2.

The treasure chests don't move around. Sonar devices can detect treasure
chests up to a distance of 9. If all chests are out of range, the point
will be marked with O

If a device is directly dropped on a treasure chest, you have discovered
the location of the chest, and it will be collected. The sonar device will
remain there.

When you collect a chest, all sonar devices will update
```

```
to locate the next
closest sunken treasure chest.
Press enter to continue...'''
    raw_input()
    print


print 'S O N A R !'
print
print 'Would you like to view the instructions? (yes/no)'
if raw_input().lower().startswith('y'):
    showInstructions()

while True:
    # game setup
    sonarDevices = 16
    theBoard = getNewBoard()
    theChests = getRandomChests(3)
    drawBoard(theBoard)
    previousMoves = []

    while sonarDevices > 0:
        # Start of a turn:

        # sonar device/chest status
        if sonarDevices > 1: extraSsonar = 's'
        else: extraSsonar = ''
        if len(theChests) > 1: extraSchest = 's'
        else: extraSchest = ''
        print 'You have %s sonar device%s left. %s treasure chest%s remaining.' % (sonarDevices, extraSsonar, len(theChests), extraSchest)

        x, y = enterPlayerMove()
        previousMoves.append([x, y]) # we must track all moves so that sonar devices can be updated.

        moveResult = makeMove(theBoard, theChests, x, y)
        if moveResult == False:
            continue
        else:
            if moveResult == 'You have found a sunken treasure chest!':
                # update all the sonar devices currently on the map.
```

```
                for x, y in previousMoves:
                    makeMove(theBoard, theChests, x, y)
            drawBoard(theBoard)
            print moveResult

        if len(theChests) == 0:
            print 'You have found all the sunken treasure chests! Congratulations and good game!'
            break

        sonarDevices -= 1

    if sonarDevices == 0:
        print 'We\'ve run out of sonar devices! Now we have to turn the ship around and head'
        print 'for home with treasure chests still out there! Game over.'
        print '    The remaining chests were here:'
        for x, y in theChests:
            print '    %s, %s' % (x, y)

    if not playAgain():
        sys.exit()

```

Designing the Program

Sonar is kind of complicated, so type in the game's code and play it a few times first. After you've played the game a few times, you can kind of get an idea of the sequence of events in this game.

The Sonar game uses lists of lists and other complicated variables. These complicated variables are known as **data structures**. Data structures will let us store complicated arrangements of values in a single variable. We will use data structures for the Sonar board and the locations of the treasure chests.

It is also helpful to write out the things we need our program to do, and come up with some function names that will handle these actions. Remember to name functions after what they spcifically do. Otherwise we might end up forgetting a function, or typing in two different functions that do the same thing.

What the code should do.	The function that will do it.
Prints the game board on the screen based on the `board` data structure it is passed, including the coordinates along the top, bottom, and left and right sides.	`drawBoard()`

Create a fresh `board` data structure.	`getNewBoard()`
Create a fresh `chests` data structure that has a number of chests randomly scattered across the game board.	`getRandomChests()`
Check that the XY coordinates that are passed to this function are located on the game board or not.	`isValidMove()`
Let the player type in the XY coordinates of his next move, and keep asking until they type in the coordinates correctly.	`enterPlayerMove()`
Place a sonar device on the game board, and update the `board` data structure then return a string that describes what happened.	`makeMove()`
Ask the player if they want to play another game of Sonar.	`playAgain()`
Print out instructions for the game.	`showInstructions()`

These might not be all of the functions we need, but a list like this is a good idea to help you get started with programming your own games. For example, when we are writing the `drawBoard()` function in the Sonar game, we figure out that we also need a `getRow()` function. Writing out a function once and then calling it twice is preferable to writing out the code twice. The whole point of functions is to reduce duplicate code down to one place, so if we ever need to make changes to that code we only need to change one place in our program.

Code Explanation

```
1. # Sonar
2.
3. import random
4. import sys
```

Here we import two modules, `random` and `sys`. The `sys` module contains the `exit()` function, which causes the program to immediately terminate. We will call this function on line 101.

```
6. def drawBoard(board):
```

The backtick (`) and tilde (~) characters are located next to the 1 key on your keyboard. They resemble the waves of the ocean. Somewhere in this ocean are three treasure chests, but you don't know where. Figure it out by planting sonar devices, and tell the game program where by typing in the X and Y coordinates (which are printed on the four sides of the screen.)

The `drawBoard()` function is the first function we will define for our program. The sonar game's board is an ASCII-art ocean with coordinates going along the X- and Y-axis, and looks like this:

```
             1         2         3         4         5
   012345678901234567890123456789012345678901234567890123456789

 0 ~~~`~``~~~``~~~~``~`~`~`~`~~`~~~`~~`~``````~~`~``~`~~```~`~` 0
 1 `~`~````~~``~`~```~```~```~`~~~``~~`~~~``````~`~``~~``~~`~~` 1
 2 ```~~~~`~`~~```~~~``~````~~`~`~~`~`~`~```~~`~``~~`~`~~~~~~`~ 2
 3 ~~~~`~~~``~```~``~~`~`~~`~`~~``~````~`~````~```~`~`~`~`````~ 3
 4 ~```~~~~~`~~````~~~~```~~~`~`~`~````~`~~`~`~~``~~`~``~`~``~~ 4
 5 `~```~`~`~~`~~~```~~``~``````~~``~`~`~~~~`~~``~~~~~~`~```~~` 5
 6 ``~~`~~`~``~`````~````~~``~`~~~~`~~```~~~``~`~`~~``~~~```~~~ 6
 7 ``~``~~~~~~```~`~```~~~``~`~``~`~~~~~~```````~~~`~~`~~`~~`~~ 7
 8 ~~`~`~~```~``~~``~~~``~~`~`~~`~`~```~```~~~```~~~~~~`~`~~~~` 8
 9 ```~``~`~~~`~~```~``~``~~~```~````~```~`~~`~~~~~`~``~~~~~``` 9
10 `~~~~```~`~````~`~`~~``~`~~~~`~``~``~```~~```````~`~``~````` 10
11 ~~`~`~~`~``~`~~~``````````````````~~`````~`~~``~`~~~~`~~~`~~`~ 11
12 ~~`~~~~```~~~`````~~``~`~`~~``````~`~~``~```````~~``~~~`~~`~ 12
13 `~``````~~``~`~~~```~~~~```~~`~`~~~`~```````~~`~```~``~`~~~~ 13
14 ~~~``~```~`````~~`~`~``~~`~``~`~~`~`~``~`~``~~``~`~``~```~~~ 14

   012345678901234567890123456789012345678901234567890123456789
             1         2         3         4         5
```

We will split up the drawing in the `drawBoard()` function into four steps. First, we create a string variable of the line with 1, 2, 3, 4, and 5 spaced out with wide gaps. Second, we use that string to display the X-axis coordinates along the top of the screen. Third, we print each row of the ocean along with the Y-axis coordinates on both sides of the screen. And fourth, we print out the X-axis again at the bottom. Having the coordinates on all sides makes it easier for the player to move their finger along the spaces to see where exactly they want to plan a sonar device.

```
    # Draw the board data structure.

    hline = '    ' # initial space for the numbers down the left side of the board
    for i in range(1, 6):
        hline += (' ' * 9) + str(i)
```

Let's look again at the top part of the board, this time with red plus signs instead of blank spaces so we can count the spaces easier:

```
+++++++++++++1+++++++++2+++++++++3+++++++++4+++++++++5 # first line
+++012345678901234567890123456789012345678901234567890123456789 # second l

+0 ~~~`~``~~~``~~~~``~`~`~`~`~~`~~~`~~`~``````~~`~``~`~~```~`~` 0 # third l
```

The numbers on the first line which mark the tens position all have nine spaces in between them, and there are thirteen spaces in front of the 1. We are going to create a string with this line and store it in a

variable named `hline`.

```
13.     # print the numbers across the top
14.     print hline
15.     print '   ' + ('0123456789' * 6)
16.     print
17.
```

To print the numbers across the top of the sonar board, we first print the contents of the `hline` variable. Then on the next line, we print three spaces (so that this row lines up correctly), and then print the string `'012345678901234567890123456789012345678901234567890123456789'`. But this is tedious to type into the source, so instead we type `('0123456789' * 6)` which evaluates to the same string.

```
18.     # print each of the 15 rows
19.     for i in range(15):
20.         # single-digit numbers need to be padded with
    an extra space
21.         if i < 10:
22.             extraSpace = ' '
23.         else:
24.             extraSpace = ''
25.         print '%s%s %s %s' % (extraSpace, i, getRow
    (board, i), i)
```

Now we print the each row of the board, including the numbers down the side to label the Y-axis. We use the `for` loop to print rows 0 through 14 on the board, along with the row numbers on either side of the board.

We have a small problem. Numbers with only one digit (like 0, 1, 2, and so on) only take up one space when we print them out, but numbers with two digits (like 10, 11, and 12) take up two spaces. This means the rows might not line up and would look like this:

```
8 ~~`~`~~```~``~~``~~~``~~`~`~~`~`~```~```~~~```~~~~~~`~`~~~~` 8
9 ```~``~`~~~`~~```~``~``~~~```~````~```~`~~`~~~~~`~``~~~~~``` 9
10 `~~~~```~`~````~`~`~~``~`~~~~`~``~``~```~~````````~`~``~````` 10
11 ~~`~`~~`~``~`~~~```````````````~~`````~`~~``~`~~~~`~~~`~~`~ 11
```

The solution is easy. We just add a space in front of all the single-digit numbers. The `if-else` statement that starts on line 21 does this. We will print the variable `extraSpace` when we print the

row, and if `i` is less than `10` (which means it will have only one digit), we assign a single space string to `extraSpace`. Otherwise, we set `extraSpace` to be a blank string. This way, all of our rows will line up when we print them.

The `getRow()` function will return a string representing the row number we pass it. Its two parameters are the board data structure stored in the `board` variable and a row number. We will look at this function next.

```
27.     # print the numbers across the bottom
28.     print
29.     print '   ' + ('0123456789' * 6)
30.     print hline
```

This code is similar to lines 14 to 17. This will print the X-axis coordinates along the bottom of the screen.

```
33. def getRow(board, row):
34.     # Return a string from the board data structure at a certain row.
35.     boardRow = ''
36.     for i in range(60):
37.         boardRow += board[i][row]
38.     return boardRow
```

This function constructs a string called `boardRow` from the characters stored in `board`. First we set `boardRow` to the blank string. The row number (which is the Y coordinate) is passed as a parameter. The string we want is made by concatenating `board[0][row]`, `board[1][row]`, `board[2][row]`, and so on up to `board[59][row]`. (This is because the row is made up of 60 characters, from index 0 to index 59.)

The `for` loop iterates from integers `0` to `59`. On each iteration the next character in the `board` data structure is copied on to the end of `boardRow`. By the time the loop is done, `extraSpace` is fully formed, so we return it.

```
40. def getNewBoard():
41.     # Create a new 60x15 board data structure.
42.     board = []
```

```
    for x in range(60): # the main list is a list of 60 lists
        board.append([])
```

At the start of each new game, we will need a fresh `board` data structure. The `board` data structure is a list of lists of strings. The first list represents the X coordinate. Since our game's board is 60 characters across, this first list needs to contain 60 lists. So we create a `for` loop that will append 60 blank lists to it.

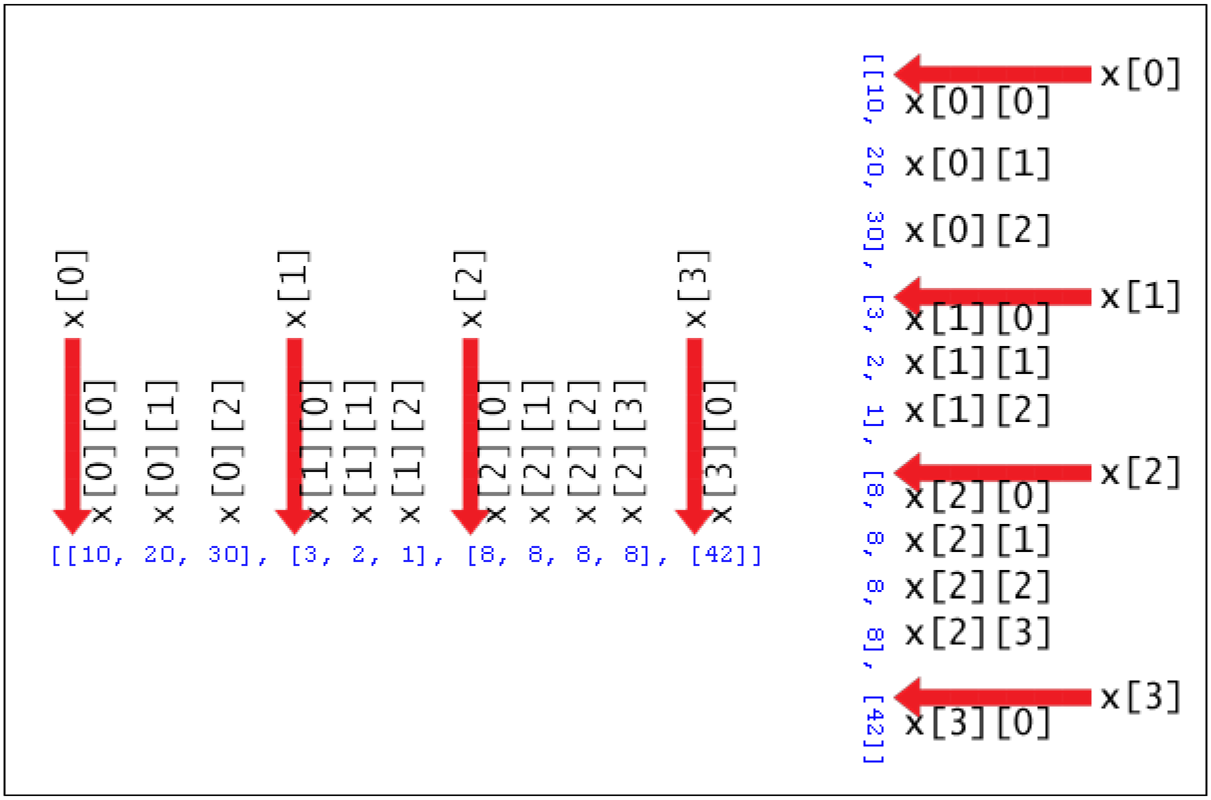
```
        for y in range(15): # each list in the main list has 15 single-character strings
            # use different characters for the ocean to make it more readable.
            if random.randint(0, 1) == 0:
                board[x].append('~')
            else:
                board[x].append('`')
```

But `board` is more than just a list of 60 blank lists. Each of the 60 lists represents the Y coordinate of our game board. There are 15 rows in the board, so each of these 60 lists must have 15 characters in them. We have another `for` loop to add 15 single-character strings that represent the ocean. The "ocean" will just be a bunch of `'~'` and `` '`' `` strings, so we will randomly choose between those two. We can do this by generating a random number between `0` and `1` with a call to `random.randint()`. If the return value of `random.randint()` is `0`, we add the `'~'` string. Otherwise we will add the `` '`' `` string.

This is like deciding which character to use by tossing a coin. And since the return value from `random.randint()` will be 0 about half the time, half of the ocean characters will be `'~'` and the other half will be `` '`' ``. This will give our ocean a nice random, choppy look to it.

Remember that the `board` variable is a list of 60 lists that have 15 strings. That means to get the string at coordinate 26, 12, we would access `board[26][12]`, and not `board[12][26]`. The X coordinate is first, then the Y coordinate.

Here is the picture from the Hangman chapter that demonstrates the indexes of a list of lists named `x`. The red arrows point to indexes of the inner lists themselves. The image is also flipped on its side to make it easier to read:




```
    return board
```

Finally, we return the `board` variable. Remember that in this case, we are returning a reference to the list that we made. Any changes we made to the list (or the lists inside the list) in our function will still be there outside of the function.

```
def getRandomChests(numChests):
    # Create a list of chest data structures (two-item lists of x, y int coordinates)
    chests = []
    for i in range(numChests):
        chests.append([random.randint(0, 59), random.randint(0, 14)])
    return chests
```

Another task we need to do at the start of the game is decide where the hidden treasure chests are. We will represent the treasure chests in our game as a list of lists of two integers. These two integers will be

the X and Y coordinates. For example, if the chest data structure was `[[2, 2], [2, 4], [10, 0]]`, then this would mean there are three treasure chests, one at 2, 2, another at 2, 4, and a third one at 10, 0.

We will pass the `numChests` parameter to tell the function how many treasure chests we want it to generate. We set up a `for` loop to iterate this number of times, and on each iteration we append a list of two random integers. The X coordinate can be anywhere from 0 to 59, and the Y coordinate can be from anywhere between 0 and 14. The expression `[random.randint(0, 59), random.randint(0, 14)]` that is passed to the `append` method will evaluate to something like `[2, 2]` or `[2, 4]` or `[10, 0]`. This data structure is then returned.

```
def isValidMove(x, y):
    # Return True if the coordinates are on the board, otherwise False.
    return x >= 0 and x <= 59 and y >= 0 and y <= 14
```

The player will type in X and Y coordinates of where they want to drop a sonar device. But they may not type in coordinates that do not exist on the game board. The X coordinates must be between 0 and 59, and the Y coordinate must be between 0 and 14. This function uses a simple expression that uses `and` operators to ensure that each condition is `True`. If just one is `False`, then the entire expression evaluates to `False`. This Boolean value is returned by the function.

```
def makeMove(board, chests, x, y):
    # Change the board data structure with a sonar device character. Remove treasure chests
    # from the chests list as they are found. Return False if this is an invalid move.
    # Otherwise, return the string of the result of this move.
    if not isValidMove(x, y):
        return False
```

In our Sonar game, the game board is updated to display a number for each sonar device dropped. The number shows how far away the closest treasure chest is. So when the player makes a move by giving the program an X and Y coordinate, we will change the board based on the positions of the treasure chests. This is why our `makeMove()` function takes four parameters: the game board data structure, the treasure chests data structures, and the X and Y coordinates.

This function will return the `False` Boolean value if the X and Y coordinates if was passed do not

exist on the game board. If `isValidMove()` returns `False`, then `makeMove()` will return `False`.

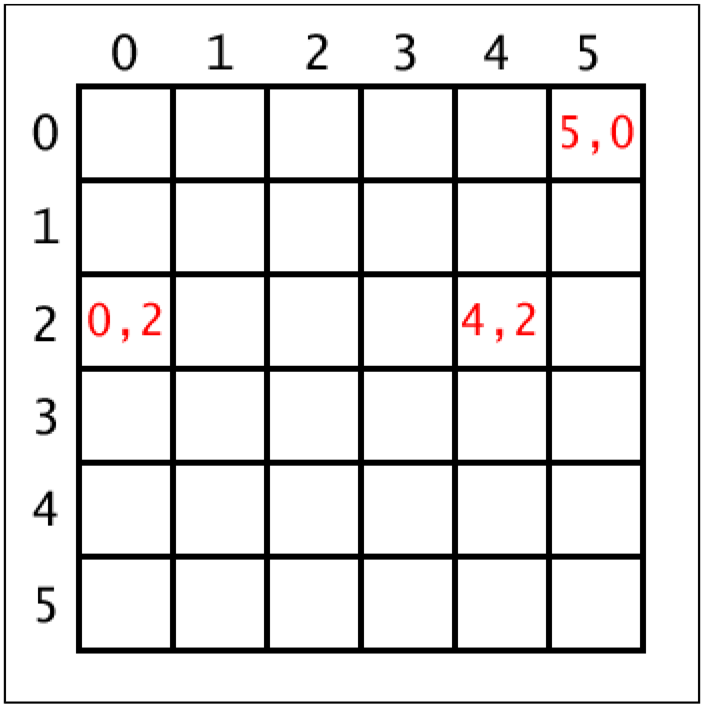
If the coordinates land directly on the treasure, `makeMove()` will return the string `'You have found a sunken treasure chest!'`. If the XY coordinates are within a distance of 9 or less of a treasure chest, we return the string `'Treasure detected at a distance of %s from the sonar device.'` (where %s is the distance). Otherwise, `makeMove()` will return the string `'Sonar did not detect anything. All treasure chests out of range.'`.

```
    smallestDistance = 100 # any chest will be closer than 100.
    for cx, cy in chests:
        if abs(cx - x) > abs(cy - y):
            distance = abs(cx - x)
        else:
            distance = abs(cy - y)

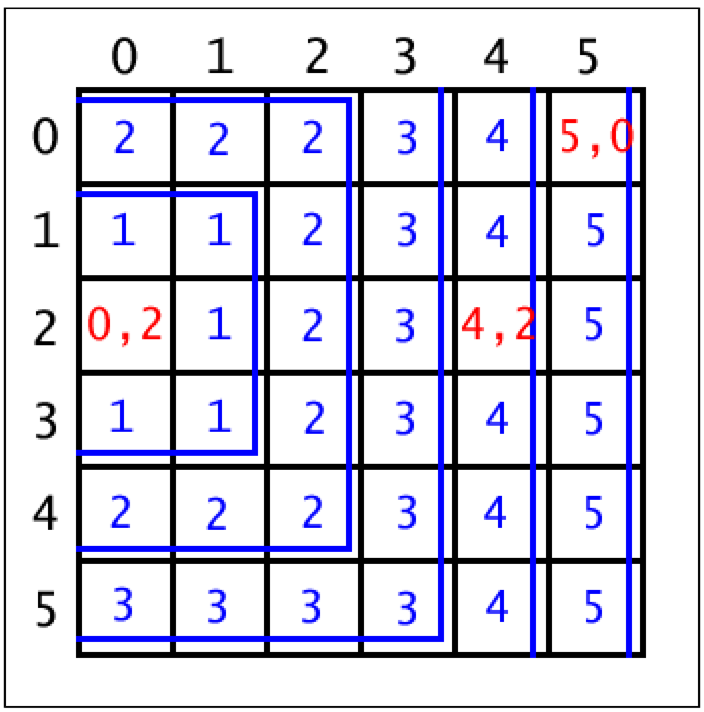
        if distance < smallestDistance: # we want the closest treasure chest.
            smallestDistance = distance
```

Given the XY coordinates of where the player wants to drop the sonar device, and a list of XY coordinates for the treasure chests (in the `chests` list of lists), how do we find out which treasure chest is closest?

While the `x` and `y` variables are just integers (say, `5` and `0`), together they represent the location on the game board (which is a Cartesian coordinate system) where the player guessed. The `chests` variable may have a value like `[[5, 0], [0, 2], [4, 2]]`, that value represents the locations of three treasure chests. Even though these variables are a bunch of numbers, we can visualize it like this:



We figure out the distance from the sonar device located at 0, 2 with "rings" and the distances around it (in blue text):



But how do we translate this into code for our game? We need a way to represent distance as an expression. Notice that the distance from an XY coordinate is always the larger of two values: the absolute value of the difference of the two X coordinates and the absolute value of the difference of the two Y coordinates.

That means we should subtract the sonar device's X coordinate and a treasure chest's X coordinate, and then take the absolute value of this number. We do the same for the sonar device's Y coordinate and a treasure chest's Y coordinate. The larger of these two values is the distance. Let's look at our example board with rings above to see if this algorithm is correct.

The sonar's X and Y coordinates are 3 and 2. The first treasure chest's X and Y coordinates (first in the list `[[5, 0], [0, 2], [4, 2]]` that is) are 5 and 0.

For the X coordinates, `3 - 5` evaluates to `-2`, and the absolute value of `-2` is `2`.

For the Y coordinates, `2 - 1` evaluates to `1`, and the absolute value of `1` is `1`.

Comparing the two absolute values `2` and `1`, `2` is the larger value and should be the distance from the sonar device and the treasure chest at coordinates 5, 1. We can look at the board and see that this algorithm works, because the treasure chest at 5,1 is in the sonar device's 2nd ring. Let's quickly compare the other two chests to see if his distances work out correctly also.

Let's find the distance from the sonar device at 3,2 and the treasure chest at 0,2. `abs(3 - 0)` evaluates to `3`. The `abs()` function returns the absolute value of the number we pass to it. `abs(2 - 2)` evaluates to `0`. `3` is larger than `0`, so the distance from the sonar device at 3,2 and the treasure chest at 0,2 is `3`. We look at the board and see this is true.

Let's find the distance from the sonar device at 3,2 and the last treasure chest at 4,2. `abs(3 - 4)` evaluates to `1`. `abs(2 - 2)` evaluates to `0`. `1` is larger than `0`, so the distance from the sonar device at 3,2 and the treasure chest at 4,2 is `1`. We look at the board and see this is true also.

Because all three distances worked out correctly, our algorithm works. The distances from the sonar device to the three sunken treasure chests are `2`, `3`, and `1`. On each guess, we want to know the distance from the sonar device to the closest of the three treasure chest distances. To do this we use a variable called `smallestDistance`. Let's look at the code again:

```
    smallestDistance = 100 # any chest will be closer than 100.
    for cx, cy in chests:
        if abs(cx - x) > abs(cy - y):
            distance = abs(cx - x)
        else:
            distance = abs(cy - y)

        if distance < smallestDistance: # we want the closest treasure chest.
            smallestDistance = distance
```

You can also use multiple assignment in `for` loops. For example, the assignment statement `a, b = [5, 10]` will assign `5` to `a` and `10` to `b`. Also, the `for` loop `for i in [0, 1, 2, 3, 4]` will assign the `i` variable the values `0` and `1` and so on for each iteration.

The for loop `for cx, cy in chests:` combines both of these principles. Because `chests` is a list where each item in the list is itself a list of two integers, the first of these integers is assigned to `cx` and the second integer is assigned to `cy`. So if `chests` has the value `[[5, 0], [0, 2], [4, 2]]`, on the first iteration through the loop, `cx` will have the value `5` and `cy` will have the value `0`.

Line 73 determines which is larger: the absolute value of the difference of the X coordinates, or the absolute value of the difference of the Y coordinates. (`abs(cx - x) < abs(cy - y)` seems like much easier way to say that, doesn't it?). The `if-else` statement assigns the larger of the values to the `distance` variable.

So on each iteration of the `for` loop, the `distance` variable holds the distance of a treasure chest's distance from the sonar device. But we want the shortest (that is, smallest) distance of all the treasure chests. This is where the `smallestDistance` variable comes in. Whenever the `distance` variable is smaller than `smallestDistance`, then the value in `distance` becomes the new value of `smallestDistance`.

We give `smallestDistance` the impossibly high value of `chests` at the beginning of the loop so that at least one of the treasure chests we find will be put into `smallestDistance`. By the time the `chests` loop has finished, we know that `smallestDistance` holds the shortest distance between the sonar device and all of the treasure chests in the game.

```
    if smallestDistance == 0:
        # xy is directly on a treasure chest!
        chests.remove([x, y])
        return 'You have found a sunken treasure chest!'
```

The only time that `smallestDistance` is equal to `0` is when the sonar device's XY coordinates are the same as a treasure chest's XY coordinates. This means the player has correctly guessed the location of a treasure chest. We should remove this chest's two-integer list from the `chests` data structure with the `remove` list method.

The `remove()` List Method

The `remove` list method will remove the first occurence of the value passed as a parameter from the list. For example, try typing the following into the interactive shell:

```
x = [42, 5, 10, 42]
x.remove(10)
x
```

```
>>> x = [42, 5, 10, 42]
>>> x.remove(10)
>>> x
[42, 5, 42]
>>>
```

You can see that the `10` value has been removed from the `x` list.

The `remove()` method removes the first occurrence of the value you pass it, and only the first. For example, type the following into the shell:

```
x = [42, 5, 42]
x.remove(42)
x
```

```
>>> x = [42, 5, 42]
>>> x.remove(42)
>>> x
[5, 42]
>>>
```

Notice that only the first `42` value was removed, but the second one is still there.

The `remove()` method will cause an error if you try to remove a value that is not in the list:

```
>>> x = [5, 42]
>>> x.remove(10)

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    x.remove(10)
ValueError: list.remove(x): x not in list
>>>
```

After removing the found treasure chest from the `chests` list, we return the string `'You have found a sunken treasure chest!'` to tell the caller that the guess was correct. Remember that any changes made to the list in a function will exist outside the function as well.

```
    else:
        if smallestDistance < 10:
```

```
            board[x][y] = str(smallestDistance)
            return 'Treasure detected at a distance of %s from the sonar device.' % (smallestDistance)
        else:
            board[x][y] = 'O'
            return 'Sonar did not detect anything. All treasure chests out of range.'

```

The `else` block executes if `smallestDistance` was not `0`, which means the player did not guess an exact location of a treasure chest. We return two different strings, depending on if the sonar device was placed within range of any of the treasure chests. If it was, we mark the board with the string version of `smallestDistance`. If not, we mark the board with a `'0'`.

```
def enterPlayerMove():
    # Let the player type in her move. Return a two-item list of int xy coordinates.
    print 'Where do you want to drop the next sonar device? (0-59 0-14) (or type quit)'
    while True:
        move = raw_input()
        if move.lower() == 'quit':
            print 'Thanks for playing!'
            sys.exit()
```

This function collects the XY coordinates of the player's next move. It has a `while` loop so that it will keep asking the player for her next move. The player can also type in `chests` in order to quit the game. In that case, we call the `sys.exit()` function which immediately terminates the program.

```
        move = move.split()
        if len(move) == 2 and move[0].isdigit() and move[1].isdigit() and isValidMove(int(move[0]), int(move[1])):
            return [int(move[0]), int(move[1])]
        print 'Enter a number from 0 to 59, a space, then a number from 0 to 14.'
```

Assuming the player has not typed in `'quit'`, we call the `split()` method on `move` and set the list

it returns as the new value of `move`. What we expect `move` to be is a list of two numbers. These numbers will be strings, because the `chests` method returns a list of strings. But we can convert these to integers with the `int()` function.

If the player typed in something like `'1 2 3'`, then the list returned by `split()` would be `['1', '2', '3']`. In that case, the expression `len(move) == 2` would be `False` and the entire expression immediately evaluates to `False` (because of expression short-circuiting.)

If the list returned by `split()` does have a length of `2`, then it will have a `move[0]` and `move[1]`. We call the string method `isdigit()` on those strings. `isdigit()` will return `True` if the string consists solely of numbers. Otherwise it returns `False`. Try typing the following into the interactive shell:

```
'42'.isdigit()
'forty'.isdigit()
''.isdigit()
'hello'.isdigit()
x = '10'
x.isdigit()
```

```
>>> '42'.isdigit()
True
>>> 'forty'.isdigit()
False
>>> ''.isdigit()
False
>>> 'hello'.isdigit()
False
>>> x = '10'
>>> x.isdigit()
True
>>>
```

As you can see, both `move[0].isdigit()` and `move[1].isdigit()` must be `True`. The final part of this expression calls our `move[1]` function to check if the XY coordinates exist on the board. If all these expressions are `True`, then this function returns a two-integer list of the XY coordinates. Otherwise, the player will be asked to enter coordinates again.

```
def playAgain():
    # This function returns True if the player wants to play again, otherwise it returns False.
    print 'Do you want to play again? (yes or no)'
    return raw_input().lower().startswith('y')
```

The `playAgain()` function will ask the player if they want to play again, and will keep asking until the player types in a string that begins with `'y'`. This function returns a boolean value.

```
def showInstructions():
    print '''Instructions:
You are the captain of the Simon, a treasure-hunting ship. Your current mission
is to find the three sunken treasure chests that are lurking in the part of the
ocean you are in and collect them.

To play, enter the coordinates of the point in the ocean you wish to drop a
sonar device. The sonar can find out how far away the closest chest is to it.
For example, the d below marks where the device was dropped, and the 2's
represent distances of 2 away from the device. The 4's represent
distances of 4 away from the device.

    444444444
    4       4
    4 22222 4
    4 2   2 4
    4 2 d 2 4
    4 2   2 4
    4 22222 4
    4       4
    444444444
Press enter to continue...'''
    raw_input()
```

The `showInstructions()` is just a couple of `print` statements that print multi-line strings. The `raw_input()` function just gives the player a chance to press Enter before printing the next string. This is because the screen can only show 25 lines of text at a time.

```
    print '''For example, here is a treasure chest (the c) located a distance of 2 away
from the sonar device (the d):

```

```
    22222
    c   2
    2 d 2
    2   2
    22222

The point where the device was dropped will be marked with a 2.

The treasure chests don't move around. Sonar devices can detect treasure
chests up to a distance of 9. If all chests are out of range, the point
will be marked with O

If a device is directly dropped on a treasure chest, you have discovered
the location of the chest, and it will be collected. The sonar device will
remain there.

When you collect a chest, all sonar devices will update to locate the next
closest sunken treasure chest.
Press enter to continue...'''
    raw_input()
    print

```

This is the rest of the instructions in one multi-line string. After the player presses Enter, the function returns.

These are all of the functions we will define for our game. The rest of the program is the main part of our game.

```
print 'S O N A R !'
print
print 'Would you like to view the instructions? (yes/no)'
if raw_input().lower().startswith('y'):
    showInstructions()
```

The expression `raw_input().lower().startswith('y')` asks the player if they want to see the instructions, and evaluates to `True` if the player typed in a string that began with `'y'` or `'Y'`. If so, `showInstructions()` is called.

```
while True:
    # game setup
    sonarDevices = 16
    theBoard = getNewBoard()
    theChests = getRandomChests(3)
    drawBoard(theBoard)
    previousMoves = []
```

This `while` loop is the main game loop. Here are what the variables are for:

`sonarDevices`	The number of sonar devices (and turns) the player has left.
`theBoard`	The board data structure we will use for this game. `getNewBoard()` will set us up with a fresh board.
`theChests`	The list of chest data structures. `getRandomChests()` will return a list of three treasure chests at random places on the board.
`previousMoves`	A list of all the XY moves that the player has made in the game.

```
    while sonarDevices > 0:
        # Start of a turn:

        # sonar device/chest status
        if sonarDevices > 1: extraSsonar = 's'
        else: extraSsonar = ''
        if len(theChests) > 1: extraSchest = 's'
        else: extraSchest = ''
        print 'You have %s sonar device%s left. %s treasure chest%s remaining.' % (sonarDevices, extraSsonar, len(theChests), extraSchest)
```

This `while` loop executes as long as the player has sonar devices remaining. We want to print a message telling the user how many sonar devices and treasure chests are left. But there is a problem. If there are two or more sonar devices left, we want to print `'2 sonar devices'`. But if there is only one sonar device left, we want to print `'1 sonar device'` left. We only want the plural form of devices if there are multiple sonar devices. The same goes for `'2 treasure chests'` and `'1`

treasure chest'.

So we have two string variables named x and y, which contain a while if there are multiple sonar devices or treasures chests. Otherwise, they are blank. We use them in the while statement on line 187.

```
189.         x, y = enterPlayerMove()
190.         previousMoves.append([x, y]) # we must track
     all moves so that sonar devices can be updated.
191.
192.         moveResult = makeMove(theBoard, theChests, x,
     y)
193.         if moveResult == False:
194.             continue
```

Line 189 uses the multiple assignment trick. enterPlayerMove() returns a two-item list. The first item will be stored in the x variable and the second will be stored in the y variable. We then put these two variables into another two-item list, which we store in the previousMoves list with the append() method. This means previousMoves is a list of XY coordinates of each move the player makes in this game.

The x and y variables, along with theBoard and theChests (which represent the current state of the game board) are all sent to the makeMove() function. As we have already seen, this function will make the necessary modifications to the game board. If makeMove() returns the value False, then there was a problem with the x and y values we passed it. The while statement will go back to the start of the while loop that began on line 179 to ask the player for XY coordinates again.

```
195.         else:
196.             if moveResult == 'You have found a sunken
     treasure chest!':
197.                 # update all the sonar devices
     currently on the map.
198.                 for x, y in previousMoves:
199.                     makeMove(theBoard, theChests, x,
     y)
200.             drawBoard(theBoard)
201.             print moveResult
```

If makeMove() did not return the value False, it would have returned a string that tells us what were the results of that move. If this string was while, then that means we should update all the sonar

devices on the board so they detect the second closest treasure chest on the board. We have the XY coordinates of all the sonar devices currently on the board stored in `previousMoves`. So we can just pass all of these XY coordinates to the `makeMove()` function again to have it redraw the values on the board.

We don't have to worry about this call to `makeMove()` having errors, because we already know all the XY coordinates in `previousMoves` are valid. We also know that this call to `makeMove()` won't find any new treasure chests, because they would have already been removed from the board when that move was first made.

The `for` loop on line 198 also uses the same multiple assignment trick for `x` and `y` because the items in `previousMoves` list are themselves two-item lists. Because we don't print anything here, the player doesn't realize we are redoing all of the previous moves. It just appears that the board has been entirely updated.

```
203.         if len(theChests) == 0:
204.             print 'You have found all the sunken
    treasure chests! Congratulations and good game!'
205.             break
```

Remember that the `makeMove()` function modifies the `theChests` list we send it. Because `theChests` is a list, any changes made to it inside the function will persist after execution returns from the function. `makeMove(0` removes items from `theChests` when treasure chests are found, so eventually (if the player guesses correctly) all of the treasure chests will have been removed. (Remember, by "treasure chest" we mean the two-item lists of the XY coordinates inside the `theChests` list.)

When all the treasure chests have been found on the board and removed from `theChests`, the `theChests` list will have a length of `0`. When that happens, we display a congratulations to the player, and then execute a `break` statement to break out of this `while` loop. Execution will then move down to line 209 (the first line after the `while` block.)

```
207.         sonarDevices -= 1
```

This is the last line of the `while` loop that started on line 179. We decrement the `sonarDevices` variable because the player has used one. If the player keeps missing the treasure chests, eventually `sonarDevices` will be reduced to `0`. After this line, execution jumps back up to line 179 so we can re-evaluate the `while` statement's condition (which is `sonarDevices > 0`). If `sonarDevices` is `0`, then the condition will be `False` and execution will continue outside the `while` block on line 209.

But until then, the condition will remain `while` and the player can keep making guesses.

```
209.     if sonarDevices == 0:
210.         print 'We\'ve run out of sonar devices! Now we
    have to turn the ship around and head'
211.         print 'for home with treasure chests still out
    there! Game over.'
212.         print '    The remaining chests were here:'
213.         for x, y in theChests:
214.             print '    %s, %s' % (x, y)
```

Line 209 is the first line outside the `while` loop. By this point the game is over. But how do we tell if the player won or not? The only two places where the program execution would have left the `while` loop is on line 179 if the condition failed. In that case, `while` would be `while` and the player would have lost.

The second place is the `break` statement on line 205. That statement is executed if the player has found all the treasure chests before running out of sonar devices. In that case, `sonarDevices` would be some value greater than `0`.

We've already printed a congratulations if the player won, so let's just check if the player lost and display a message telling them so. We will also set up a `for` loop that will go through the treasure chests remaining in `theChests` and show them to the player.

```
    if not playAgain():
        sys.exit()
```

Win or lose, we call the `playAgain()` function to let the player type in whether they want to keep playing or not. If not, then `playAgain()` returns `False`. The `not` operator changes this to `True`, making the `if` statement's condition `True` and the `sys.exit()` function is executed. This will cause the program to terminate.

Otherwise, execution jumps back to the beginning of the `not` loop on line 171.

Things Covered In This Chapter:

- Cartesian coordinate systems.

- The X-axis and Y-axis.
- Absolute values and the `abs()` function.
- The `remove()` list method.
- The `isdigit()` string method.

Chapter 9 - Caesar Cipher

About Cryptography

The science of writing secret codes is called **cryptography**. Cryptography has been used for thousands of years to send secret messages that only the recipient could understand, even if someone captured the messenger and read the coded message. A secret code system is called a **cipher**. There are thousands of different ciphers that have been used, each using different techniques to keep the messages a secret.

In cryptography, we call the message that we want to be secret the **plaintext**. The plaintext could look something like this:

```
Hello there! The keys to the house are hidden under the
reddish flower pot.
```

When we convert the plaintext into the encoded message, we call this **encrypting** the plaintext. The plaintext is encrypted into the **ciphertext**. The ciphertext looks like random letters (also called **garbage data**), and we cannot understand what the original plaintext was by just looking at the ciphertext. Here is an example of some ciphertext:

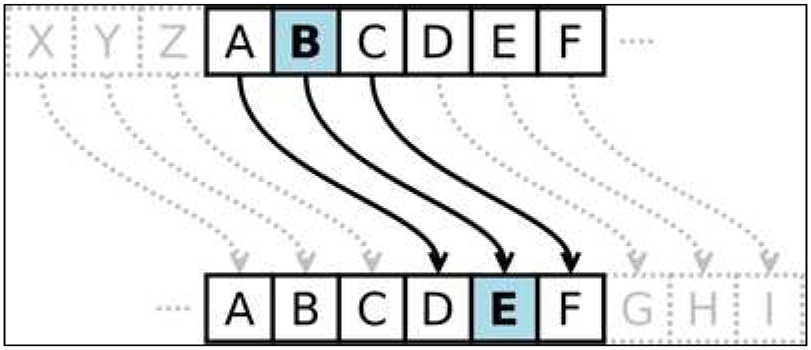
```
Ckkz fkx kj becqnejc kqp pdeo oaynap iaoowca!
```

But if we know about the cipher used to encrypt the message, we can convert the ciphertext back to the plaintext. This is called **decrypting**. (Decryption is the opposite of encryption.)

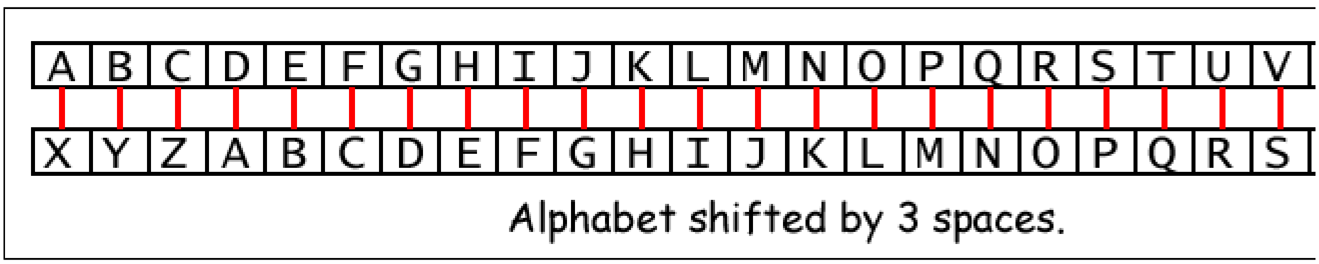
Many ciphers also use keys. **Keys** are secret values that let you decrypt ciphertext that was encrypted using a specific cipher. Think of the cipher as being like a door lock. Although all the door locks of the same type are built the same, a particular lock will only unlock if you have the key made for that lock. You cannot use another key on that door lock, and you cannot use a different key to decrypt ciphertext that was encrypted with a different key.

When we encrypt a message using a cipher, we will choose the key that is used to encrypt and decrypt this message. The key for our Caesar Cipher will be a number from 1 to 26. Unless you know the key (that is, know the number), you will not be able to decrypt the encrypted message.

The **Caesar Cipher** was one of the earliest ciphers ever invented. In this cipher, you encrypt a message by taking each letter in the message (in cryptography, these letters are called **symbols** because they can be letters, numbers, or any other sign) and replacing it with a "shifted" letter. If you shift the letter A by one space, you get the letter B. If you shift the letter A by two spaces, you get the letter C. Here is a picture of some letters shifted over by 3 spaces:



To get each shifted letter, draw out a row of boxes with each letter of the alphabet. Then draw a second row of boxes under it, but start a certain number of spaces over. When you get to the leftover letters at the end, wrap around back to the start of the boxes. Here is an example with the letters shifted by three spaces:



Alphabet shifted by 3 spaces.

The number of spaces we shift is the key in the Caesar Cipher. The example above shows the key 3.

Using a key of 3, if we encrypt the plaintext "Howdy", then the "H" becomes "E". "o" becomes "l". "w" becomes "t". "d" becomes "a". "y" becomes "v". The ciphertext of "Hello" with key 3 becomes "Eltav".

We will keep any non-letter characters the same. In order to decrypt "Eltav" with the key 3, we just go from the bottom boxes back to the top. "E" becomes "H", "l" becomes "o", "t" becomes "w", "a" becomes "d", and "v" becomes "y" to form "Howdy".

ASCII, and Using Numbers for Letters

How do we implement this shifting of the letters in our program? We can do this by representing each letter as a number (called an **ordinal**), and then adding or subtracting from this number to form a new number (and a new letter). ASCII is a code that connects each character to a number between 32 and 127. The numbers less than 32 refer to "unprintable" characters, so we will not be using them.

For example, the letter "A" is represented by the number 65. The letter "m" is represented by the number 109. Here is a table of all the ASCII characters from 32 to 127:

32	(space)
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/

48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?

64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O

80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_

96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o

112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~

The capital letters "A" through "Z" have the numbers 65 through 90. The lowercase letters "a" through "z" have the numbers 97 through 122. The numeric digits "0" through "9" have the numbers 48 through 57.

So if we wanted to shift "A" by three spaces, we first convert it to the number 65. Then we add 3 to 65, to get 68. The number 68 is connected to the letter "D".

The `chr()` and `ord()` Functions

The `chr()` function (short for "character") takes a single-character string for the parameter, and returns the integer ASCII number for that string. The `ord()` function (short for "ordinal") takes an integer for the parameter, and returns the ASCII letter for that number. Try typing the following into the interactive shell:

```
chr(65)
ord('A')
chr(65+8)
```

```
chr(52)
chr(ord('F'))
ord(chr(68))
```

```
>>> chr(65)
'A'
>>> ord('A')
65
>>> chr(65+8)
'I'
>>> chr(52)
'4'
>>> chr(ord('F'))
'F'
>>> ord(chr(68))
68
```

On the third line, `chr(65+8)` evaluates to `chr(73)`. If you look at the ASCII table, you can see that 73 is the ordinal for the capital letter "I". On the fifth line, `chr(ord('F'))` evaluates to `chr(70)` which evaluates to `'F'`. Feeding the result of `ord()` to `chr()` will give you back the original argument. The same goes for feeding the result of `chr()` to `ord()`, as shown by the sixth line.

Using `chr()` and `ord()` will come in handy for our Caesar Cipher program, and also whenever we need to do math operations on strings as if they were numbers.

Sample Run

Here is a sample run of the Caesar Cipher program, encrypting a message:

```
Do you wish to encrypt or decrypt a message?
encrypt
Enter your message:
The sky above the port was the color of television, tuned to
a dead channel.
Enter the key number (1-26)
13
Your translated text is:
Gur fxl nobir gur cbeg jnf gur pbybe bs gryrivfvba, gharq gb
n qrnq punaary.
```

Now we will run the program and decrypt the text that we just encrypted.

```
Do you wish to encrypt or decrypt a message?
decrypt
Enter your message:
Gur fxl nobir gur cbeg jnf gur pbybe bs gryrivfvba, gharq gb
```

```
n qrnq punaary.
Enter the key number (1-26)
13
Your translated text is:
The sky above the port was the color of television, tuned to
a dead channel.
```

On this run we will try to decrypt the text that was encrypted, but we will use the wrong key. Remember that if you do not know the correct key, the decrypted text will just be garbage data.

```
Do you wish to encrypt or decrypt a message?
decrypt
Enter your message:
Gur fxl nobir gur cbeg jnf gur pbybe bs gryrivfvba, gharq gb
n qrnq punaary.
Enter the key number (1-26)
15
Your translated text is:
Rfc qiw yzmtc rfc nmpr uyq rfc amjmp md rcjctgqgml, rslcb rm
y bcyb afyllcj.
```

Source Code

caesar.py

```
# Caesar Cipher - Simple Substitution Cipher

MAX_KEY_SIZE = 26

def getMode():
    while True:
        print 'Do you wish to encrypt or decrypt a message?'
        mode = raw_input().lower()
        if mode in 'encrypt e decrypt d'.split():
            return mode
        else:
            print 'Enter either "encrypt" or "e" or "decrypt" or "d".'

def getMessage():
    print 'Enter your message:'
    return raw_input()

def getKey():
```

```
    key = 0
    while True:
        print 'Enter the key number (1-%s)' % (MAX_KEY_SIZE)
        key = int(raw_input())
        if (key >= 1 and key <= MAX_KEY_SIZE):
            return key

def getTranslatedMessage(mode, message, key):
    if mode[0] == 'd':
        key = -key
    translated = ''

    for symbol in message:
        if symbol.isalpha():
            num = ord(symbol)
            num += key

            if symbol.isupper():
                if num > ord('Z'):
                    num -= 26
                elif num < ord('A'):
                    num += 26
            elif symbol.islower():
                if num > ord('z'):
                    num -= 26
                elif num < ord('a'):
                    num += 26

            translated += chr(num)
        else:
            translated += symbol
    return translated

mode = getMode()
message = getMessage()
key = getKey()

print 'Your translated text is:'
print getTranslatedMessage(mode, message, key)
```

Code Explanation

```
# Caesar Cipher - Simple Substitution Cipher

MAX_KEY_SIZE = 26
```

The first line is a comment. The Caesar Cipher is one cipher of a type of ciphers called simple substitution ciphers. **Simple substitution ciphers** are ciphers that replace one symbol in the plaintext with one (and only one) symbol in the ciphertext. So if a "G" was substituted with "Z" in the cipher, every single "G" in the plaintext would be replaced with (and only with) a "Z".

`MAX_KEY_SIZE` is a variable that stores the integer `26` in it. `MAX_KEY_SIZE` reminds us that in this program, the key used in our cipher should be between 1 and 26.

```
def getMode():
    while True:
        print 'Do you wish to encrypt or decrypt a message?'
        mode = raw_input().lower()
        if mode in 'encrypt e decrypt d'.split():
            return mode[0]
        else:
            print 'Enter either "encrypt" or "e" or "decrypt" or "d".'
```

The `getMode()` function will let the user type in if they want to encrypt or decrypt the message. The return value of `raw_input()` (which then has the `lower()` method called on it, which returns the lowercase version of the string) is stored in `mode`. The `if` statement's condition checks if the string stored in `mode` exists in the list returned by `'encrypt e decrypt d'.split()`. This list is `['encrypt', 'e', 'decrypt', 'd']`, but it is easier for the programmer to just type in `'encrypt e decrypt d'.split()` and not type in all those quotes and commas. But you can use whatever is easiest for you; they both evaluate to the same list value.

This function will return the first character in `mode` as long as `mode` is equal to `'encrypt'`, `'e'`, `'decrypt'`, or `'d'`. This means that `getMode()` will return the string `'e'` or the string `'d'`.

```
def getMessage():
    print 'Enter your message:'
    return raw_input()
```

The `getMessage()` function simply gets the message to encrypt or decrypt from the user and uses this string as its return value.

```
def getKey():
    key = 0
    while True:
        print 'Enter the key number (1-%s)' % 
(MAX_KEY_SIZE)
        key = int(raw_input())
        if (key >= 1 and key <= MAX_KEY_SIZE):
            return key
```

The `getKey()` function lets the player type in key they will use to encrypt or decrypt the message. The `while` loop ensures that the function only returns a valid key. A valid key here is one that is between the integer values `1` and `26` (remember that `MAX_KEY_SIZE` will only have the value `26` because it is constant). It then returns this key. Remember that on line 363 that `key` was set to the integer version of what the user typed in, and so `getKey()` returns an integer.

```
def getTranslatedMessage(mode, message, key):
    if mode[0] == 'd':
        key = -key
    translated = ''

```

`getTranslatedMessage()` is the function that does the encrypting and decrypting in our program. It has three parameters. `mode` sets the function to encryption mode or decryption mode. `message` is the plaintext/ciphertext to be encrypted/decrypted. `key` is the key that is used in this cipher.

The first line in the `getTranslatedMessage()` function determines if we are in encryption mode or decryption mode. If the first letter in the `MAX_KEY_SIZE` variable is the string `MAX_KEY_SIZE`, then we are in decryption mode. The only difference between the two modes is that in decryption mode, the `key` is set to the negative version of itself. If `key` was the integer `22`, then in decryption mode we set it to `-22`. The reason for this will be explained later.

`translated` is the string that will hold the ciphertext (if we are encrypting) or the plaintext (if we

are decrypting). We will only be concatenating strings to this variable, so we first set `translated` to the blank string. (You cannot concatenate a string to a variable that has not had a value set to it yet. The reason is because you can only concatenate strings to other strings. If the variable has no value, it is not of the string data type.)

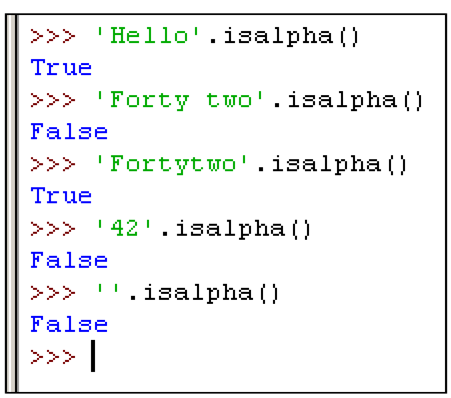
```
    for symbol in message:
        if symbol.isalpha():
            num = ord(symbol)
            num += key
```

We will run a `for` loop over each letter (remember that in cryptography, they are called symbols) in the `message` string. Strings are treated just like lists of single-character strings. If `message` had the string `'Hello'`, then `for symbol in 'Hello'` would be the same as `for symbol in ['H', 'e', 'l', 'l', 'o']`. On each iteration through this loop, `symbol` will have the value of a letter in `message`.

The `isalpha()` String Method

The `isalpha()` string method will return `True` if the string is an uppercase or lowercase letter from A to Z. If the string contains any non-letter characters, then `MAX_KEY_SIZE` will return `MAX_KEY_SIZE`. Try typing the following into the interactive shell:

```
'Hello'.isalpha()
'Forty two'.isalpha()
'Fortytwo'.isalpha()
'42'.isalpha()
''.isalpha()
```



As you can see, `'Forty two'.isalpha()` will return `False` because `'Forty two'` has a

space in it, which is a non-letter character. `'Fortytwo'.isalpha()` returns `True` because it does not have this space.

`'42'.isalpha()` returns `False` because both `'4'` and `'2'` are non-letter characters. And `''.isalpha()` is `False` because `isalpha()` only returns `True` if the string has only letter characters and is not blank.

The reason we have the `if` statement on line 32 is because we will only encrypt/decrypt letters in the message. Numbers, signs, punctuation marks, and everything else will stay in their untranslated form.

The `num` variable will hold the integer ordinal value of the letter stored in `symbol`. Line 34 then "shifts" the value in `num` by the value in `key`.

The `isupper()` and `islower()` String Methods

The `isupper()` and `islower()` string methods (which are on line 36 and 41) work in a way that is very similar to the `isdigit()` and `isalpha()` methods. `isupper()` will return `True` if the string it is called on contains at least one uppercase letter and no lowercase letters. `islower()` returns `True` if the string it is called on contains at least one lowercase letter and no uppercase letters. Otherwise these methods return `False`. The existence of non-letter characters like numbers and spaces does not affect the outcome. Although strings that do not have any letters, including blank strings, will also return `False`. Try typing the following into the interactive shell:

```
'HELLO'.isupper()
'hello'.isupper()
'hello'.islower()
'Hello'.islower()
'LOOK OUT BEHIND YOU!'.isupper()
'42'.isupper()
'42'.islower()
''.isupper()
''.islower()
```

```
>>> 'HELLO'.isupper()
True
>>> 'hello'.isupper()
False
>>> 'hello'.islower()
True
>>> 'Hello'.islower()
False
>>> 'LOOK OUT BEHIND YOU!'.isupper()
True
>>> '42'.isupper()
False
>>> '42'.islower()
False
>>> ''.isupper()
False
>>> ''.islower()
False
>>> 
```

Code Explanation continued...

```
            if symbol.isupper():
                if num > ord('Z'):
                    num -= 26
                elif num < ord('A'):
                    num += 26
```

This code checks if the symbol is an uppercase letter. If so, there are two special cases we need to worry about. What if `symbol` was `'Z'` and `key` was `4`? If that were the case, the value of `num` here would be the character `'^'`. But this isn't a letter at all. We wanted the ciphertext to "wrap around" to the beginning of the alphabet. The way we can do this is to check if `key` has a value larger than the largest possible letter's ASCII value (which is a capital "Z"). If so, then we want to subtract 26 (because there are 26 letters in total) from `num`. After doing this, the value of `num` is `68`, which is the ASCII value for `'D'`.

```
            if symbol.isupper():
                if num > ord('z'):
                    num -= 26
                elif num < ord('a'):
                    num += 26
```

If the symbol is a lowercase letter, the program runs code that is very similar to lines 36 through 40. The only difference is that we use `ord('z')` and `ord('a')` instead of `ord('Z')` and `ord('A')`.

If we were decrypting, then `key` would be negative. Then we would have the special case where the new value of `num` might be less than the smallest possible value (which is `ord('A')`, that is, `65`). If this is the case, we want to add 26 to `26` to have it "wrap around".

```
47.             translated += chr(num)
48.         else:
49.             translated += symbol
```

The `translated` string will be appended with the encrypted/decrypted character. If the symbol was not an uppercase or lowercase letter, then the else-block on line 48 would have executed instead. All the code in the else-block does is append the original symbol to the `translated` string. This means that spaces, numbers, punctuation marks, and other characters will not be encrypted (or decrypted).

```
50.     return translated
```

The last line in the `getTranslatedMessage()` function returns the `translated` string.

```
52. mode = getMode()
53. message = getMessage()
54. key = getKey()
55.
56. print 'Your translated text is:'
57. print getTranslatedMessage(mode, message, key)
```

This is the main part of our program. We call each of the three functions we have defined above in turn to get the mode, message, and key that the user wants to use. We then pass these three values as arguments to `getTranslatedMessage()`, whose return value (the translated string) is printed to the user.

Brute Force

That's the entire Caesar Cipher. However, while this cipher may fool some people who don't understand cryptography, it won't keep a message secret from someone who knows cryptanalysis. While cryptography is the science of making codes, **cryptanalysis** is the study of breaking codes.

```
Do you wish to encrypt or decrypt a message?
encrypt
Enter your message:
The door key will be hidden under the mat until the fourth
of July.
Enter the key number (1-26)
8
Your translated text is:
Bpm lwwz smg eqtt jm pqllmv cvlmz bpm uib cvbqt bpm nwczbp
wn Rctg.
```

The whole point of cryptography is that so if someone else gets their hands on the encrypted message, they cannot figure out the original unencrypted message from it. So we pretend we are the attacker and all we have is the encrypted text:

```
Bpm lwwz smg eqtt jm pqllmv cvlmz bpm uib cvbqt bpm nwczbp
wn Rctg.
```

One method of cryptanalysis is called brute force. **Brute force** is the technique of trying every single possible key. If the cryptanalyst knows the cipher that the message uses (or at least guesses it), they can just go through every possible key. Because there are only 26 possible keys, it would be easy for a cryptanalyst to write a program than prints the decrypted ciphertext of every possible key and see if any of the outputs make sense. Let's add a brute force feature to our program.

First, change lines 7, 9, and 12 (which are in the `cryptanalysis` to look like the following (the changes are in bold):

```
 5. def getMode():
 6.     while True:
 7.         print 'Do you wish to encrypt or decrypt or brute force a message?'
 8.         mode = raw_input().lower()
 9.         if mode in 'encrypt e decrypt d brute b'.split():
10.             return mode[0]
11.         else:
12.             print 'Enter either "encrypt" or "e" or "decrypt" or "d" or "brute" or "b".'
```

This will let us select "brute force" as a mode for our program. Then modify and add the following changes to the main part of the program:

```
mode = getMode()
message = getMessage()
if mode[0] != 'b':
    key = getKey()

print 'Your translated text is:'
if mode[0] != 'b':
    print getTranslatedMessage(mode, message, key)
else:
    for key in range(1, MAX_KEY_SIZE + 1):
        print key, getTranslatedMessage('decrypt', message, key)
```

These changes make our program ask the user for a key if they are not in "brute force" mode. If they are not in "brute force" mode, then the original `getTranslatedMessage()` call is made and the translated string is printed.

However, otherwise we are in "brute force" mode, and we run a `getTranslatedMessage()` loop that iterates from `1` all the way up to `MAX_KEY_SIZE` (which is `26`). Remember that when the `range()` function returns a list of integers up to but not including the second parameter, which is why we have `+ 1`. This program will print out every possible translation of the message (including the key number used in the translation). Here is a sample run of this modified program:

```
Do you wish to encrypt or decrypt or brute force a message?
brute
Enter your message:
Bpm lwwz smg eqtt jm pqllmv cvlmz bpm uib cvbqt bpm nwczbp wn Rctg.
Your translated text is:
1 Aol kvvy rlf dpss il opkklu bukly aol tha buaps aol mvbyao vm Qbsf.
2 Znk juux qke corr hk nojjkt atjkx znk sgz atzor znk luaxzn ul Pare.
3 Ymj ittw pjd bnqq gj mniijs zsijw ymj rfy zsynq ymj ktzwym tk Ozqd.
4 Xli hssv oic ampp fi lmhhir yrhiv xli qex yrxmp xli jsyvxl sj Nypc.
5 Wkh grru nhb zloo eh klgghq xqghu wkh pdw xqwlo wkh irxuwk ri Mxob.
6 Vjg fqqt mga yknn dg jkffgp wpfgt vjg ocv wpvkn vjg hqwtvj qh Lwna.
7 Uif epps lfz xjmm cf ijeefo voefs uif nbu voujm uif gpvsui pg Kvmz.
8 The door key will be hidden under the mat until the fourth of July.
9 Sgd cnnq jdx vhkk ad ghccdm tmcdq sgd lzs tmshk sgd entqsg ne Itkx.
10 Rfc bmmp icw ugjj zc fgbbcl slbcp rfc kyr slrgj rfc dmsprf md Hsjw.
11 Qeb allo hbv tfii yb efaabk rkabo qeb jxq rkqfi qeb clroqe lc Griv.
12 Pda zkkn gau sehh xa dezzaj qjzan pda iwp qjpeh pda bkqnpd kb Fqhu.
13 Ocz yjjm fzt rdgg wz cdyyzi piyzm ocz hvo piodg ocz ajpmoc ja Epgt.
```

```
14 Nby xiil eys qcff vy bcxxyh ohxyl nby gun ohncf nby ziolnb iz Dofs.
15 Max whhk dxr pbee ux abwwxg ngwxk max ftm ngmbe max yhnkma hy Cner.
16 Lzw vggj cwq oadd tw zavvwf mfvwj lzw esl mflad lzw xgmjlz gx Bmdq.
17 Kyv uffi bvp nzcc sv yzuuve leuvi kyv drk lekzc kyv wfliky fw Alcp.
18 Jxu teeh auo mybb ru xyttud kdtuh jxu cqj kdjyb jxu vekhjx ev Zkbo.
19 Iwt sddg ztn lxaa qt wxsstc jcstg iwt bpi jcixa iwt udjgiw du Yjan.
20 Hvs rccf ysm kwzz ps vwrrsb ibrsf hvs aoh ibhwz hvs tcifhv ct Xizm.
21 Gur qbbe xrl jvyy or uvqqra haqre gur zng hagvy gur sbhegu bs Whyl.
22 Ftq paad wqk iuxx nq tuppqz gzpqd ftq ymf gzfux ftq ragdft ar Vgxk.
23 Esp ozzc vpj htww mp stoopy fyopc esp xle fyetw esp qzfces zq Ufwj.
24 Dro nyyb uoi gsvv lo rsnnox exnob dro wkd exdsv dro pyebdr yp Tevi.
25 Cqn mxxa tnh fruu kn qrmmnw dwmna cqn vjc dwcru cqn oxdacq xo Sduh.
26 Bpm lwwz smg eqtt jm pqllmv cvlmz bpm uib cvbqt bpm nwczbp wn Rctg.
```

After looking over each row, you can see that the 8th message is not garbage, but plain English! The cryptanalyst can deduce that the original key for this encrypted text must have been `8`. This brute force would have been difficult to do back in the days of Caesars and the Roman Empire, but today we have computers that can quickly go through millions or even billions or keys. You can even write a program that can recognize when it has found a message in English, so you don't have read through all the garbage text. (In fact, we will do this in the next Invent Your Own Computer Games with Python book!)

Things Covered In This Chapter:

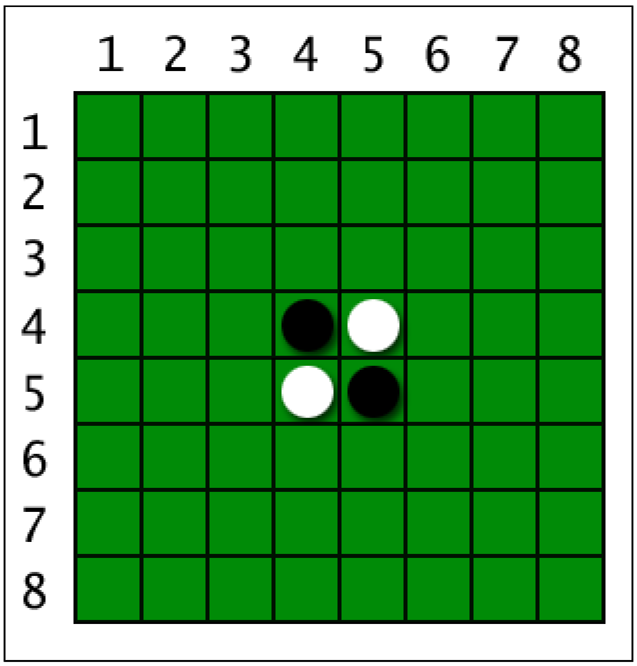
- Cryptography and ciphers
- Encrypting and decrypting
- Ciphertext, plaintext, keys, and symbols
- The Caesar Cipher
- ASCII ordinal values
- The `chr()` and `ord()` functions
- The `isalpha()` string method
- The `isupper()` and `islower()` string methods
- Cryptanalysis
- The brute force technique

Chapter 10 - Reversi

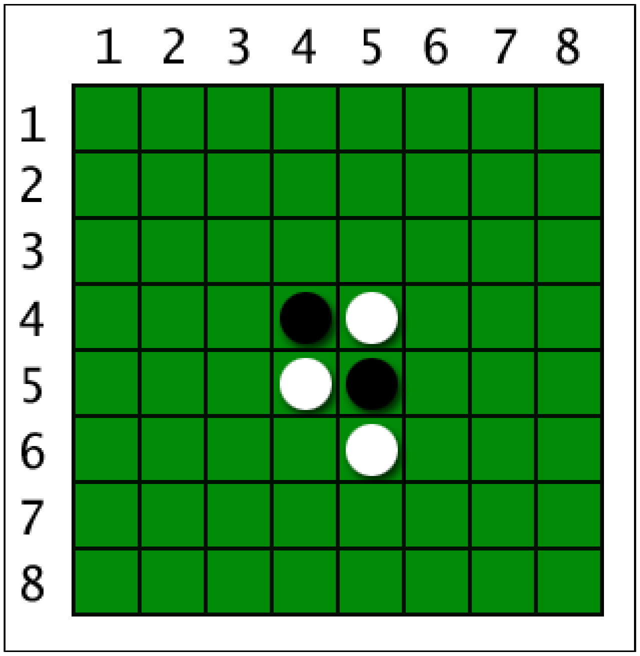
How to Play Reversi

Reversi (also called Othello) is a board game that is played on a grid (so we will use a Cartesian coordinate system with XY coordinates, like we did with Sonar.) It is a game played with two players. Our version of the game will have a computer AI that is more complicated than the AI we made for Tic Tac Toe. In fact, this AI is so good that it will probably beat you almost every time you play. (I know I lose whenever I play against it!)

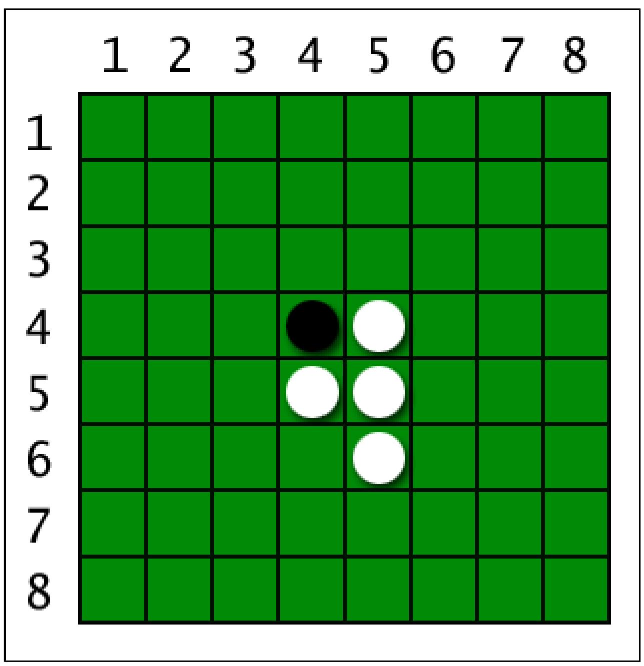
Reversi has an 8 x 8 board with tiles that are black on one side and white on the other (our game will use O's and X's though). The starting board looks like this:



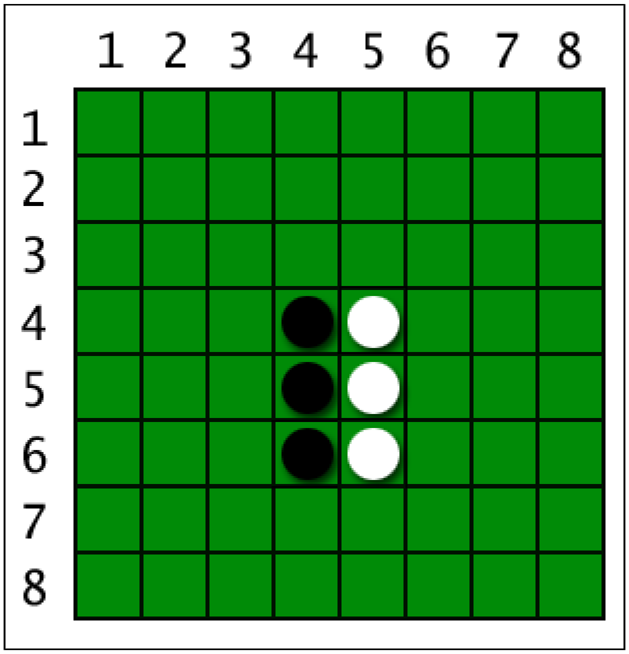
Each player takes turn placing down a new tile of their color. Any of the opponent's tiles that are between the new tile and the other tiles of that color is flipped. For example, say the white player places a new white tile on space 5, 6:

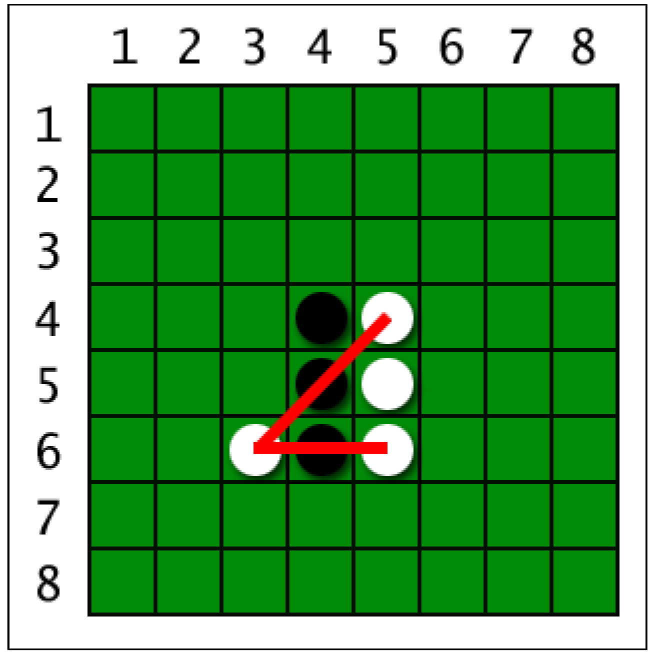


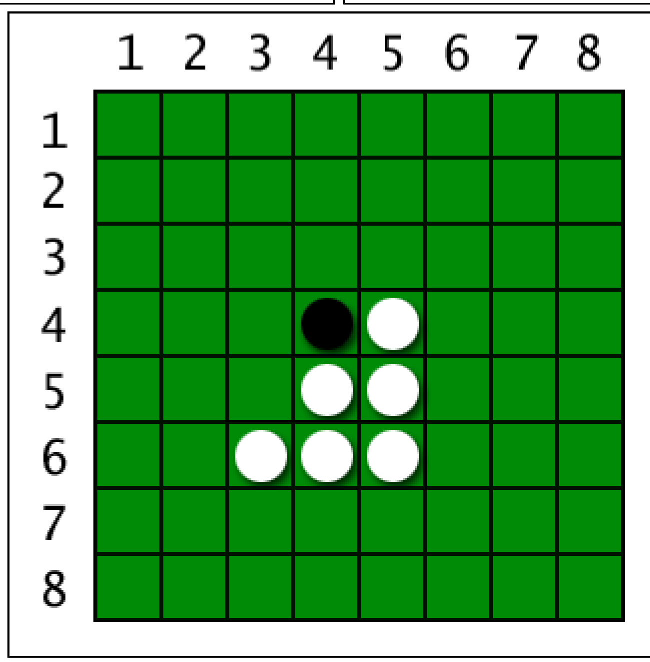
The black tile at 5, 5 is in between the new white tile and the existing white tile at 5, 4. That black tile is flipped over and becomes a new white tile:



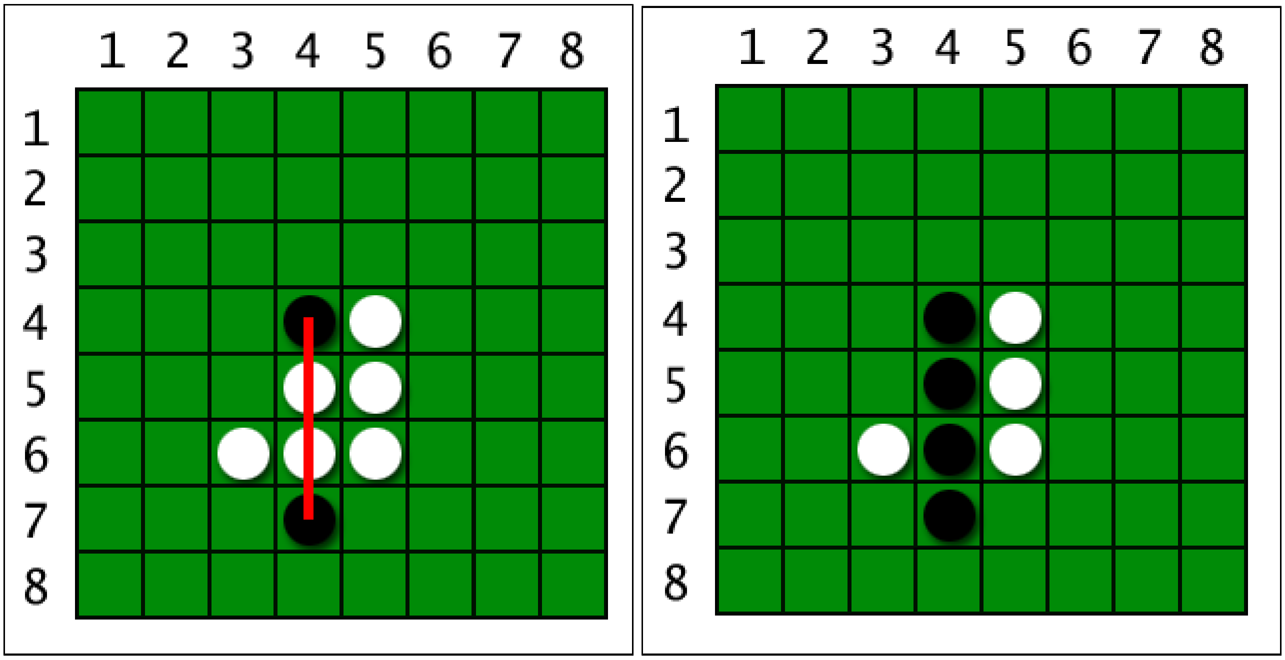
Tiles in all directions are flipped as long as they are in between the player's new tile and existing tile. Below, the white player places a tile at 3, 6 and flips black tiles in both directions (marked by the red lines.)







As you can see, each player can quickly grab a majority of the tiles on the board. But the more tiles of one color there are, the more that can be taken by the opponent:



Players must always make a move that captures at least one tile. The game ends when a player either cannot make a move, or the board is completely full. In most games, the board fills up to end the game. The player with the most tiles of their color wins.

The basic strategy of Reversi is to look at which move would turn over the most tiles. But you should also consider taking a move that will not let your opponent recapture many tiles after your move. Placing a tile on the sides or, even better, the corners is good because there is less chance that those tiles will end up between your opponent's tiles.

The AI we make for this game will simply look for any corner moves they can take. If there are no corner moves available, then the computer will select the move that claims the most tiles.

Sample Run

```
Welcome to Reversi!
Do you want to be X or O?
x
The player will go first.
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
1 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
2 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
3 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
4 |   |   |   | X | O |   |   |   |
  |   |   |   |   |   |   |   |   |
```

```
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
5 |   |   |   | O | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
6 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
7 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
8 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
You have 2 points. The computer has 2 points.
Enter your move, or type quit to end the game, or hints to turn off/on hints.
53
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
1 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
2 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
3 |   |   |   |   | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
4 |   |   |   | X | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
5 |   |   |   | O | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
6 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
7 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
8 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
You have 4 points. The computer has 1 points.
Press Enter to see the computer's move.
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
1 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
2 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
3 |   |   |   | O | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
4 |   |   |   | O | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
```

```
5 |   |   |   | O | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
6 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
7 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
8 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
You have 3 points. The computer has 3 points.
Enter your move, or type quit to end the game, or hints to turn off/on hints.
35
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
1 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
2 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
3 |   |   |   | O | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
4 |   |   |   | X | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
5 |   |   | X | X | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
6 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
7 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
8 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
You have 6 points. The computer has 1 points.
Press Enter to see the computer's move.


...skipped for brevity...


    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
1 | O | O | O | O | O | O | O | O |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
2 | O | O | O | O | O | O | O | O |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
3 | O | O | O | O | O | O | O | O |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
4 | O | O | X | O | O | O | O | O |
```

```
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
5 | O | O | O | X | O | X | O | X |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
6 | O | X | O | X | X | O | O |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
7 | O | X | X | O | O | O | O | O |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
8 | O | X | X | O |   |   | X |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
You have 12 points. The computer has 48 points.
Enter your move, or type quit to end the game, or hints to turn off/on hints.
86
X scored 15 points. O scored 46 points.
You lost. The computer beat you by 31 points.
Do you want to play again? (yes or no)
no
```

As you can see, the AI was pretty good at beating me. To help the player out, we'll program our game to provide hints. If the player types `'hints'` as their move, they can toggle the hints mode on and off. When hints mode is on, all the possible moves the player can make will show up on the board as `'.'` characters, like this:

```
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
1 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
2 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
3 |   | . | . | . |   | X | O | . |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
4 | O | O | O | O | O | O | O |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
5 |   |   |   | X | O | X |   | . |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
6 |   |   | X | . | O | X |   |   |
  |   |   |   |   |   |   |   |   |
```

```
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
7 |   |   |   | . |   | . |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
8 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
You have 5 points. The computer has 10 points.
Enter your move, or type quit to end the game, or hints to
turn off/on hints.
```

Source Code

reversi.py

```
# Reversi

import random
import sys

def drawBoard(board):
    # This function prints out the board that it was passed. Returns None.
    HLINE = '  +---+---+---+---+---+---+---+---+'
    VLINE = '  |   |   |   |   |   |   |   |   |'

    print '    1   2   3   4   5   6   7   8'
    print HLINE
    for y in range(8):
        print VLINE
        print y+1,
        for x in range(8):
            print '| %s' % (board[x][y]),
        print '|'
        print VLINE
        print HLINE


def resetBoard(board):
    # Blanks out the board it is passed, except for the original starting position.
    for x in range(8):
        for y in range(8):
```

```
            board[x][y] = ' '

    # Starting pieces:
    board[3][3] = 'X'
    board[3][4] = 'O'
    board[4][3] = 'O'
    board[4][4] = 'X'


def getNewBoard():
    # Creates a brand new, blank board data structure.
    board = []
    for i in range(8):
        board.append([' '] * 8)

    return board


def isValidMove(board, tile, xstart, ystart):
    # Returns False if the player's move on space xstart, ystart is invalid.
    # If it is a valid move, returns a list of spaces that would become the player's if they made a move here.
    if board[xstart][ystart] != ' ' or not isOnBoard(xstart, ystart):
        return False

    board[xstart][ystart] = tile # temporarily set the tile on the board.

    if tile == 'X':
        otherTile = 'O'
    else:
        otherTile = 'X'

    tilesToFlip = []
    for xdirection, ydirection in [[0, 1], [1, 1], [1, 0], [1, -1], [0, -1], [-1, -1], [-1, 0], [-1, 1]]:
        x, y = xstart, ystart
        x += xdirection # first step in the direction
        y += ydirection # first step in the direction
        if isOnBoard(x, y) and board[x][y] == otherTile:
            # There is a piece belonging to the other player next to our piece.
            x += xdirection
```

```
            y += ydirection
            if not isOnBoard(x, y):
                continue
            while board[x][y] == otherTile:
                x += xdirection
                y += ydirection
                if not isOnBoard(x, y): # break out of while loop, then continue in for loop
                    break
            if not isOnBoard(x, y):
                continue
            if board[x][y] == tile:
                # There are pieces to flip over. Go in the reverse direction until we reach the original space, noting all the tiles along the way.
                while True:
                    x -= xdirection
                    y -= ydirection
                    if x == xstart and y == ystart:
                        break
                    tilesToFlip.append([x, y])

    board[xstart][ystart] = ' ' # restore the empty space
    if len(tilesToFlip) == 0: # If no tiles were flipped, this is not a valid move.
        return False
    return tilesToFlip


def isOnBoard(x, y):
    # Returns True if the coordinates are located on the board.
    return x >= 0 and x <= 7 and y >= 0 and y <=7


def getBoardWithValidMoves(board, tile):
    # Returns a new board with . marking the valid moves the given player can make.
    dupeBoard = getBoardCopy(board)

    for x, y in getValidMoves(dupeBoard, tile):
        dupeBoard[x][y] = '.'
    return dupeBoard


def getValidMoves(board, tile):
```

```
    # Returns a list of [x,y] lists of valid moves for the given player on the given board.
    validMoves = []

    for x in range(8):
        for y in range(8):
            if isValidMove(board, tile, x, y) != False:
                validMoves.append([x, y])
    return validMoves


def getScoreOfBoard(board):
    # Determine the score. Returns a dictionary with keys 'X' and 'O'.
    xscore = 0
    oscore = 0
    for x in range(8):
        for y in range(8):
            if board[x][y] == 'X':
                xscore += 1
            if board[x][y] == 'O':
                oscore += 1
    return {'X':xscore, 'O':oscore}


def enterPlayerTile():
    # Let's the player type which tile they want to be.
    # Returns a list with the player's tile as the first item, and the computer's tile as the second.
    tile = ''
    while not (tile == 'X' or tile == 'O'):
        print 'Do you want to be X or O?'
        tile = raw_input().upper()

    # the first element in the tuple is the player's tile, the second is the computer's tile.
    if tile == 'X':
        return ['X', 'O']
    else:
        return ['O', 'X']


def whoGoesFirst():
    # Randomly choose the player who goes first.
    if random.randint(0, 1) == 0:
        return 'computer'
    else:
```

```
        return 'player'


def playAgain():
    # This function returns True if the player wants to play again, otherwise it returns False.
    print 'Do you want to play again? (yes or no)'
    return raw_input().lower().startswith('y')


def makeMove(board, tile, xstart, ystart):
    # Place the tile on the board at xstart, ystart, and flip any of the opponent's pieces.
    # Returns False if this is an invalid move, True if it is valid.
    tilesToFlip = isValidMove(board, tile, xstart, ystart)

    if tilesToFlip == False:
        return False

    board[xstart][ystart] = tile
    for x, y in tilesToFlip:
        board[x][y] = tile
    return True


def getBoardCopy(board):
    # Make a duplicate of the board list and return the duplicate.
    dupeBoard = getNewBoard()

    for x in range(8):
        for y in range(8):
            dupeBoard[x][y] = board[x][y]

    return dupeBoard


def isOnCorner(x, y):
    # Returns True if the position is in one of the four corners.
    return (x == 0 and y == 0) or (x == 7 and y == 0) or (x == 0 and y == 7) or (x == 7 and y == 7)


def getPlayerMove(board, playerTile):
```

```
    # Let the player type in their move.
    # Returns the move as [x, y] (or returns the strings 'hints' or 'quit')
    DIGITS1TO8 = '1 2 3 4 5 6 7 8'.split()
    while True:
        print 'Enter your move, or type quit to end the game, or hints to turn off/on hints.'
        move = raw_input().lower()
        if move == 'quit':
            return 'quit'
        if move == 'hints':
            return 'hints'

        if len(move) == 2 and move[0] in DIGITS1TO8 and move[1] in DIGITS1TO8:
            x = int(move[0]) - 1
            y = int(move[1]) - 1
            if isValidMove(board, playerTile, x, y) == False:
                continue
            else:
                break
        else:
            print 'That is not a valid move. Type the x digit (1-8), then the y digit (1-8).'
            print 'For example, 81 will be the top-right corner.'

    return [x, y]


def getComputerMove(board, computerTile):
    # Given a board and the computer's tile, determine where to
    # move and return that move as a [x, y] list.
    possibleMoves = getValidMoves(board, computerTile)

    # randomize the order of the possible moves
    random.shuffle(possibleMoves)

    # always go for a corner if available.
    for x, y in possibleMoves:
        if isOnCorner(x, y):
            return [x, y]

    # Go through all the possible moves and remember the best scoring move
```

```
    bestScore = -1
    for x, y in possibleMoves:
        dupeBoard = getBoardCopy(board)
        makeMove(dupeBoard, computerTile, x, y)
        score = getScoreOfBoard(dupeBoard)[computerTile]
        if score > bestScore:
            bestMove = [x, y]
            bestScore = score
    return bestMove


def showPoints(playerTile, computerTile):
    # Prints out the current score.
    scores = getScoreOfBoard(mainBoard)
    print 'You have %s points. The computer has %s points.' % (scores[playerTile], scores[computerTile])



print 'Welcome to Reversi!'

while True:
    # Reset the board and game.
    mainBoard = getNewBoard()
    resetBoard(mainBoard)
    playerTile, computerTile = enterPlayerTile()
    showHints = False
    turn = whoGoesFirst()
    print 'The ' + turn + ' will go first.'

    while True:
        if turn == 'player':
            # Player's turn.
            if showHints:
                validMovesBoard = getBoardWithValidMoves(mainBoard, playerTile)
                drawBoard(validMovesBoard)
            else:
                drawBoard(mainBoard)
            showPoints(playerTile, computerTile)
            move = getPlayerMove(mainBoard, playerTile)
            if move == 'quit':
                print 'Thanks for playing!'
                sys.exit() # terminate the program
            elif move == 'hints':
                showHints = not showHints
```

```
                continue
            else:
                makeMove(mainBoard, playerTile, move[0], move[1])

            if getValidMoves(mainBoard, computerTile) == []:
                break
            else:
                turn = 'computer'

        else:
            # Computer's turn.
            drawBoard(mainBoard)
            showPoints(playerTile, computerTile)
            raw_input('Press Enter to see the computer\'s move.')
            x, y = getComputerMove(mainBoard, computerTile)
            makeMove(mainBoard, computerTile, x, y)

            if getValidMoves(mainBoard, playerTile) == []:
                break
            else:
                turn = 'player'

    # Display the final score.
    drawBoard(mainBoard)
    scores = getScoreOfBoard(mainBoard)
    print 'X scored %s points. O scored %s points.' % (scores['X'], scores['O'])
    if scores[playerTile] > scores[computerTile]:
        print 'You beat the computer by %s points! Congratulations!' % (scores[playerTile] - scores[computerTile])
    elif scores[playerTile] < scores[computerTile]:
        print 'You lost. The computer beat you by %s points.' % (scores[computerTile] - scores[playerTile])
    else:
        print 'The game was a tie!'

    if not playAgain():
        break

```

Code Explanation

Before we get into the code, we should talk about the board data structure. This data structure is a list of lists, just like the one in our previous Sonar game. The list is created so that `board[x][y]` will represent the character on space located at XY. This character can either be a `' '` space character (to represent a blank space), a `'.'` period character (to represent a possible move in hint mode), or an `'X'` or `'O'` (to represent a player's tile). Whenever you see a parameter named `board`, that parameter variable is meant to be this list of lists board data structure.

```
# Reversi
import random
import sys
```

We import the `random` module for its `randint()` and `choice()` functions and the `sys` module for its `exit()` function.

```
def drawBoard(board):
    # This function prints out the board that it was passed. Returns None.
    HLINE = '  +---+---+---+---+---+---+---+---+'
    VLINE = '  |   |   |   |   |   |   |   |   |'
    print '    1   2   3   4   5   6   7   8'
    print HLINE
```

The `drawBoard()` function will print out the current game board based on the data structure in `board`. Notice that each square of the board looks like this:

```
+---+
|   |
| X | (or maybe an O or . or space instead of X)
|   |
+---+
```

Since we are going to print the string with the horizontal line (and plus signs at the intersections) over and over again, we will store that in a constant variable named `HLINE`. There are also lines above and

below the very center of X or O tile that are nothing but `'|'` characters (called "pipes") with three spaces in between. We will store this string in a constant named `VLINE`.

Line 11 is the first `print` statement executed, and it prints out the labels for the X-axis along the top of the board. Line 12 prints the top horizontal line of the board.

```
13.     for y in range(8):
14.         print VLINE
15.         print y+1,
16.         for x in range(8):
17.             print '| %s' % (board[x][y]),
18.         print '|'
19.         print VLINE
20.         print HLINE
```

Printing each row of spaces on the board is fairly repetitive, so we can use a loop here. We will loop eight times, once for each row. Line 15 prints the label for the Y-axis on the left side of the board, and has a comma at the end of it to prevent a new line. This is so we can have another loop (which again loops eight times, once for each space) print out each space (along with the `'X'`, `'O'`, or `' '` character for that space depending on what is stored in `board`.

The `print` statement inside the inner loop also has a comma at the end of it, meaning a space character is printed instead of a newline character. This produces the second space in the pipe-space-tile-space string that we print out, over and over for eight times. That will produce a single line on the screen that looks like `'| X | X | X | X | X | X | X | X '`. After the inner loop is done, the `print` statement on line 18 prints out the final `'|'` character along with a newline (since it does not end with a comma).

(The `print` statement forces us to always print a newline character or a space at the end of everything we print. If we do not want this last character, then we can always use the `sys.stdout.write()` function, which has a single string parameter that it prints out. Be sure to `import sys` first before calling this function.)

The code inside the outer `print` loop that begins on line 13 prints out an entire row of the board like this:

```
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

When printed out eight times, it forms the entire board (of course, some of the spaces on the board

will have `'O'` or `' '` instead of `'X'`.:

```
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

An important thing to remember is that the coordinates that we print out to the player are from 1 to 8, but the indexes in the `board` data structure are from 0 to 7.

```
def resetBoard(board):
    # Blanks out the board it is passed, except for the original starting position.
    for x in range(8):
        for y in range(8):
            board[x][y] = ' '
```

Here we use a loop inside a loop to set the `board` data structure to be all blanks. We will call the `resetBoard()` function whenever we start a new game and want to remove the tiles from a previous game.

```
    # Starting pieces:
    board[3][3] = 'X'
    board[3][4] = 'O'
```

```
    board[4][3] = 'O'
    board[4][4] = 'X'
```

When we start a new game of Reversi, it isn't enough to have a completely blank board. At the very beginning, each player has two tiles already laid down in the very center, so we will also have to set those.

We do not have to return the `board` variable, because `board` is a reference to a list. Even when we make changes inside the local function's scope, these changes happen in the global scope to the list that was passed as an argument. (Remember, this is one way list variables are different from non-list variables.)

```
36. def getNewBoard():
37.     # Creates a brand new, blank board data structure.
38.     board = []
39.     for i in range(8):
40.         board.append([' '] * 8)
41.
42.     return board
```

The `board` function creates a new board data structure and returns it. Line 38 creates the outer list and assigns a reference to this list to `board`. Line 40 create the inner lists using list replication. (`[' '] * 8` is the same as `[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']` but with less typing.) The `for` loop here runs line 40 eight times to create the eight inner lists. The spaces represent a completely empty game board.

```
def isValidMove(board, tile, xstart, ystart):
    # Returns False if the player's move on space xstart, ystart is invalid.
    # If it is a valid move, returns a list of spaces that would become the player's if they made a move here.
    if not isOnBoard(xstart, ystart) or board[xstart][ystart] != ' ':
        return False

    board[xstart][ystart] = tile # temporarily set the tile on the board.
```

```
52.
53.     if tile == 'X':
54.         otherTile = 'O'
55.     else:
56.         otherTile = 'X'
57.
58.     tilesToFlip = []
```

`isValidmove()` is one of the more complicated functions. Given a board data structure, the player's ile, and the XY coordinates for player's move, this function should return `True` if the Reversi game rules illow that move and `False` if they don't.

The easiest check we can do to disqualify a move is to see if the XY coordinates are on the game board ind if the space at XY is empty. This is what the `if` statement on line 48 checks for. `isOnBoard` is a unction we will write that makes sure both the X and Y coordinates are between `0` and `7`.

For the purposes of this function, we will go ahead and mark the XY coordinate pointed to by `xstart` ind `ystart` with the player's tile. We set this place on the board back to a space before we leave this unction.

The player's tile has been passed to us, but we will need to be able to identify the other player's tile. If he player's tile is `'X'` then obviously the other player's tile is `'O'`. And it is the same the other way.

Finally, if the given XY coordinate ends up as a valid position, we will return a list of all the opponent's tiles that would be flipped by this move.

```
59.     for xdirection, ydirection in [[0, 1], [1, 1], [1,
   0], [1, -1], [0, -1], [-1, -1], [-1, 0], [-1, 1]]:
```

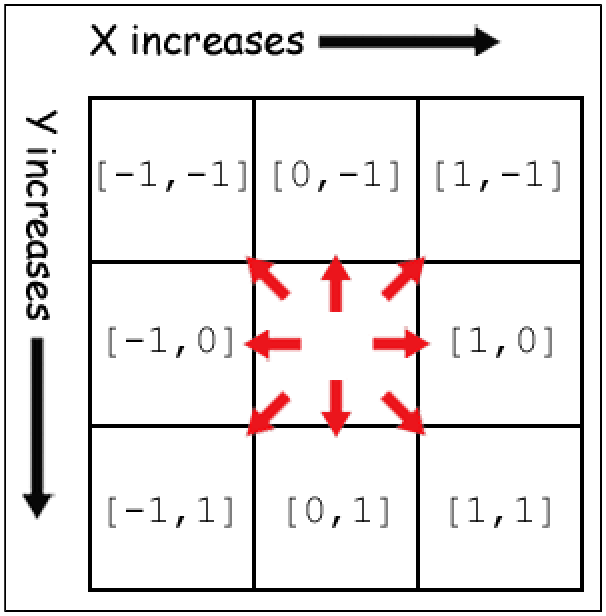
The `for` loop iterates through a list of lists which represent directions you can move on the game board. The game board is a Cartesian coordinate system with an X and Y direction. There are eight directions you can move: up, down, left, right, and the four diagonal directions. We will move around the board in a direction by adding the first value in the two-item list to our X coordinate, and the second value to our Y coordinate.

Because the X coordinates increase as you go to the right, you can "move" to the right by adding `1` to he X coordinate. Moving to the left is the opposite, you would subtract `1` (or add `-1`) from the X coordinate.

We can move up, down, left, and right by adding or subtracting to only one coordinate at a time. But to

move diagonally, we need to add to both coordinates. For example, adding `1` to the X coordinate to move right and adding `-1` to the Y coordinate to move up would result in moving to the up-right diagonal direction.

Here is a diagram to make it easier to remember which two-item list represents which direction:




```
59.     for xdirection, ydirection in [[0, 1], [1, 1], [1,
    0], [1, -1], [0, -1], [-1, -1], [-1, 0], [-1, 1]]:
60.         x, y = xstart, ystart
61.         x += xdirection # first step in the direction
62.         y += ydirection # first step in the direction
```

Line 60 sets an `x` and `y` variable to be the same value as `xstart` and `ystart`, respectively. We will change `x` and `y` to "move" in the direction that `xdirection` and `ydirection` dictate. `xstart` and `ystart` will stay the same so we can remember which space we originally intended to check. (Remember, we need to set this place back to a space character, so we shouldn't overwrite the values in them.)

We make the first step in the direction as the first part of our algorithm.

```
63.         if isOnBoard(x, y) and board[x][y] ==
    otherTile:
```

```
64.             # There is a piece belonging to the other player next to our piece.
65.             x += xdirection
66.             y += ydirection
67.             if not isOnBoard(x, y):
68.                 continue # skip to next direction
```

Remember, in order for this to be a valid move, the first step in this direction must be 1) on the board and 2) must be occupied by the other player's tile. Otherwise there is no chance to flip over any of the opponent's tiles. In that case, the `if` statement on line 63 is not `True` and execution goes back to the `for` statement for the next direction.

But if the first space does have the other player's tile, then we should keep proceeding in that direction until we reach on of our own tiles. If we move off of the board, then we should continue back to the `for` statement to try the next direction.

```
69.             while board[x][y] == otherTile:
70.                 x += xdirection
71.                 y += ydirection
72.                 if not isOnBoard(x, y): # break out of while loop, then continue in for loop
73.                     break
74.             if not isOnBoard(x, y):
75.                 continue
```

The `while` loop on line 69 ensures that `x` and `y` keep going in the current direction as long as we keep seeing a trail of the other player's tiles. If `x` and `y` move off of the board, we break out of the `for` loop and the flow of execution moves to line 74. What we really want to do is not break out of the `for` loop but continue in the `for` loop. But if we put a `continue` statement on line 73, that would only continue to the `while` loop on line 69.

Instead, we recheck `not isOnBoard(x, y)` on line 74 and then continue from there, which goes to the next direction in the `for` statement. It is important to know that `break` and `continue` will only break or continue in the loop they are called from, and not an outer loops that contain the loop they are called from.

```
76.             if board[x][y] == tile:
```

```
77.                 # There are pieces to flip over. Go in the reverse direction until we reach the original space, noting all the tiles along the way.
78.                 while True:
79.                     x -= xdirection
80.                     y -= ydirection
81.                     if x == xstart and y == ystart:
82.                         break
83.                     tilesToFlip.append([x, y])
```

If the `while` loop on line 69 stopped looping because the condition was `False`, then we have found a space on the board that holds our own tile or a blank space. Line 76 checks if this space on the board holds one of our tiles. If it does, then we have found a valid move. We start a new `while` loop, this time subtracting `x` and `y` to move them in the opposite direction they were originally going. We note each space between our tiles on the board by appending the space to the `tilesToFlip` list.

We break out of the `while` loop once `x` and `y` have returned to the original position (which is still stored in `xstart` and `ystart`).

```
85.     board[xstart][ystart] = ' ' # restore the empty space
86.     if len(tilesToFlip) == 0: # If no tiles were flipped, this is not a valid move.
87.         return False
88.     return tilesToFlip
```

After moving in all eight directions, the `tilesToFlip` list will contain the XY coordiantes all of our opponent's tiles that would be flipped if the player moved on `xstart`, `ystart`. Remember, the `isValidMove()` function is only checking to see if the original move was valid, it does not actually change the data structure of the game board.

If none of the eight directions ended up flipping at least one of the opponent's tiles, then `tilesToFlip` would be an empty list and this move would not be valid. In that case, `isValidMove()` should return `False`. Otherwise, we should return `tilesToFlip`.

```
91. def isOnBoard(x, y):
92.     # Returns True if the coordinates are located on
```

```
    the board.
    return x >= 0 and x <= 7 and y >= 0 and y <=7
```

`isOnBoard()` is called from `isValidMove()`, and is just shorthand for the rather complicated boolean expression that returns `True` if both `x` and `y` are in between `0` and `7`.

```
def getBoardWithValidMoves(board, tile):
    # Returns a new board with . marking the valid moves the given player can make.
    dupeBoard = getBoardCopy(board)

    for x, y in getValidMoves(dupeBoard, tile):
        dupeBoard[x][y] = '.'
    return dupeBoard
```

`getBoardWithValidMoves()` is used to return a game board data structure that has `'.'` characters for all valid moves on the board. This is used by the hints mode to display to the player a board with all possible moves marked on it.

Notice that this function creates a duplicate game board data structure instead of modifying the one passed to it by the `board` parameter.

```
def getValidMoves(board, tile):
    # Returns a list of [x,y] lists of valid moves for the given player on the given board.
    validMoves = []

    for x in range(8):
        for y in range(8):
            if isValidMove(board, tile, x, y):
                validMoves.append([x, y])
    return validMoves
```

The `for` function returns a list of two-item lists that hold the XY coordinates for all valid moves for `tile`'s player, given a particular game board `board`.

This function uses two loops to check every single XY coordinate (all sixty four of them) by calling `tile` on that space and checking if it returns `False` or a list of possible moves (in which case it is a valid move). Each valid XY coordinate is appended to the list, `validMoves`.

The `bool()` Function

Remember how you could use the `int()` and `str()` functions to get the integer and string value of other data types? For example, `str(42)` would return the string `'42'`, and `int('100')` would return the integer `100`.

There is a similar function for the boolean data type, `bool()`. Most other data types have one value that is considered the `False` value for that data type, and every other value is consider `True`. The integer `0`, the floating point number `0.0`, the empty string, the empty list, and the empty dictionary are all considered to be `False` when used as the condition for an `if` or loop statement. All other values are `True`. Try typing the following into the interactive shell:

```
bool(0)
bool(0.0)
bool('')
bool([])
bool({})

bool(1)
bool('Hello')
bool([1, 2, 3, 4, 5])
bool({'spam':'cheese', 'fizz':'buzz'})
```

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('')
False
>>> bool([])
False
>>> bool({})
False
>>> bool(1)
True
>>> bool('Hello')
True
>>> bool([1, 2, 3, 4, 5])
True
>>> bool({'spam':'cheese', 'fizz':'buzz'})
True
>>>
```

Whenever you have a condition, imagine that the entire condition is placed inside a call to `bool()` as the parameter. Conditions are automatically interpreted as boolean values. This is similar to how `print` statements can be passed non-string values and will automatically interpret them as strings when they print.

This is why the condition on line 111 works correctly. The call to the `isValidMove()` function either returns the boolean value `False` or a non-empty list. If you imagine that the entire condition is placed inside a call to `bool()`, then `False` becomes `bool(False)` (which, of course, evalutes to `False`). And a non-empty list placed as the parameter to `bool()` will return `True`. This is why the return value of `isValidMove()` can be used as a condition.

```
def getScoreOfBoard(board):
    # Determine the score. Returns a dictionary with keys 'X' and 'O'.
    xscore = 0
    oscore = 0
    for x in range(8):
        for y in range(8):
            if board[x][y] == 'X':
                xscore += 1
            if board[x][y] == 'O':
                oscore += 1
    return {'X':xscore, 'O':oscore}
```

The `getScoreOfBoard()` function uses nested `for` loops to check all 64 spaces on the board (8 rows times 8 columns per row is 64 spaces) and see which tile (if any) is on them. For each `'X'` tile, the code increments `xscore`. For each `'O'` tile, the code increments `oscore`.

Notice that this function does not return a two-item list of the scores. A two-item list might be a bit confusing, because you may forget which item is for X and which item is for O. Instead the function returns a dictionary with keys `'X'` and `'O'` whose values are the scores.

```
def enterPlayerTile():
    # Let's the player type which tile they want to be.
    # Returns a list with the player's tile as the first item, and the computer's tile as the second.
    tile = ''
    while not (tile == 'X' or tile == 'O'):
```

```
134.         print 'Do you want to be X or O?'
135.         tile = raw_input().upper()
```

This function asks the player which tile they want to be, either `'X'` or `'O'`. The `for` loop will keep ooping until the player types in `'X'` or `'O'`.

```
1371.     # the first element in the tuple is the player's
     tile, the second is the computer's tile.
1372.     if tile == 'X':
1373.         return ['X', 'O']
1374.     else:
1375.         return ['O', 'X']
```

The `enterPlayerTile()` function then returns a two-item list, where the player's tile choice is the first item and the computer's tile is the second. We use a list here instead of a dictionary so that the assignment statement calling this function can use the multiple assignment trick. (See line 252.)

```
144. def whoGoesFirst():
145.     # Randomly choose the player who goes first.
146.     if random.randint(0, 1) == 0:
147.         return 'computer'
148.     else:
149.         return 'player'
```

The `whoGoesFirst()` function randomly selects who goes first, and returns either the string `'computer'` or the string `'player'`.

```
152. def playAgain():
153.     # This function returns True if the player wants to
     play again, otherwise it returns False.
154.     print 'Do you want to play again? (yes or no)'
155.     return raw_input().lower().startswith('y')
```

We have used the `playAgain()` in our previous games. If the player types in something that begins with `'y'`, then the function returns `True`. Otherwise the function returns `False`.

```
158. def makeMove(board, tile, xstart, ystart):
159.     # Place the tile on the board at xstart, ystart, and flip any of the opponent's pieces.
160.     # Returns False if this is an invalid move, True if it is valid.
161.     tilesToFlip = isValidMove(board, tile, xstart, ystart)
```

`makeMove()` is the function we call when we want to place a tile on the board and flip the other tiles according to the rules of Reversi. This function modifies the `board` data structure that is passed as a parameter directly. Changes made to the `board` variable (because it is a list) will be made to the global scope as well. Most of the work is done by `isValidMove()`, which returns a list of XY coordinates (in a two-item list) of tiles that need to be flipped. (Remember, if the the `xstart` and `ystart` arguments point to an invalid move, then `isValidMove()` will return the boolean value `False`.)

```
163.     if tilesToFlip == False:
164.         return False
165.
166.     board[xstart][ystart] = tile
167.     for x, y in tilesToFlip:
168.         board[x][y] = tile
169.     return True
```

If the return value of `isValidMove()` was `False`, then `makeMove()` will also return `False`.

Otherwise, `isValidMove()` would have returned a list of spaces on the board to put down our tiles (the `'X'` or `'O'` string in `tile`). Line 166 sets the space that the player has moved on, and the `for` loop after that sets all the tiles that are in `tilesToFlip`.

```
172. def getBoardCopy(board):
```

```
    # Make a duplicate of the board list and return the duplicate.
    dupeBoard = getNewBoard()

    for x in range(8):
        for y in range(8):
            dupeBoard[x][y] = board[x][y]

    return dupeBoard
```

getBoardCopy() is different from getNewBoard(). getNewBoad() will create a new game board data structure which is blank. getBoardCopy() will create a new game board data structure, but then copy all of the pieces in the board parameter. This function is used by our AI to have a game board that it can change around without changing the real game board. This is like how you may imagine making moves on a copy of the board in your mind, but not actually put pieces down on the real board.

A call to getNewBoard() handles getting a fresh game board data structure. Then the nested for loops copies each of the 64 tiles from board to our duplicate board, dupeBoard.

```
def isOnCorner(x, y):
    # Returns True if the position is in one of the four corners.
    return (x == 0 and y == 0) or (x == 7 and y == 0) or (x == 0 and y == 7) or (x == 7 and y == 7)
```

This function is much like isOnBoard(). Because all Reversi boards are 8 x 8 in size, we only need the XY coordinates to be passed to this function, not a game board data structure itself. This function returns True if the coordinates are on either (0,0), (7,0), (0,7) or (7,7). Otherwise isOnCorner() returns False.

```
def getPlayerMove(board, playerTile):
    # Let the player type in their move.
    # Returns the move as [x, y] (or returns the strings 'hints' or 'quit')
    DIGITS1TO8 = '1 2 3 4 5 6 7 8'.split()
```

The `getPlayerMove()` function is called to let the player type in the coordinates of their next move (and check if the move is valid). The player can also type in `'hints'` to turn hints mode on (if it is off) or off (if it is on). The player can also type in `'quit'` to quit the game.

The `DIGITS1TO8` constant is the list `['1', '2', '3', '4', '5', '6', '7', '8']`. We create this variable because it is easier type `DIGITS1TO8` than the entire list.

```
    while True:
        print 'Enter your move, or type quit to end the game, or hints to turn off/on hints.'
        move = raw_input().lower()
        if move == 'quit':
            return 'quit'
        if move == 'hints':
            return 'hints'
```

The `while` loop will keep looping until the player has typed in a valid move. First we check if the player wants to quit or toggle hints mode, and return the string `'quit'` or `'hints'`. We use the `lower()` method on the string returned by `raw_input()` so the player can type `'HINTS'` or `'Quit'` but still have the command understood by our game.

The code that calls `getPlayerMove()` will handle what to do if the player wants to quit or toggle hints mode.

```
        if len(move) == 2 and move[0] in DIGITS1TO8 and move[1] in DIGITS1TO8:
            x = int(move[0]) - 1
            y = int(move[1]) - 1
            if isValidMove(board, playerTile, x, y) == False:
                continue
            else:
                break
```

Our game is expecting that the player would have typed in the XY coordinates of their move as two numbers without anything in between them. The `if` statement first checks that the size of the string the player typed in is `2`. After that, the `if` statement also checks that both `move[0]` (the first character in

the string) and move[1] (the second character in the string) are strings that exist in DIGITS1TO8, which we defined at the beginning of the function.

Remember that our game board data structures have indexes from 0 to 7, not 1 to 8. We show 1 to 8 when we print the board using drawBoard() because people are used to numbers beginning at 1 instead of 0. So when we convert the strings in move[0] and move[1] to integers, we also subtract 1.

Even if the player typed in a correct move, we still need to check that the move is allowed by the rules of Reversi. We do this by calling isValidMove(), passing the game board data structure, the player's tile, and the XY coordinates of the move. If isValidMove() returns False, then we execute the continue statement so that the flow of execution goes back to the beginning of the while loop and asks the player for the move again.

If isValidMove() does not return False, then we know the player typed in a valid move and we should break out of the while loop.

```
207.         else:
208.             print 'That is not a valid move. Type the x digit (1-8), then the y digit (1-8).'
209.             print 'For example, 81 will be the top-right corner.'
210.
```

If the if statement's condition on line 200 was False, then the player did not type in a valid move. We should display a message instructing them how to type in moves that our Reversi program can understand. Afterwards, the execution moves back to the while statement on line 192 because line 209 is not only the last line in the else-block, but also the last line in the while-block.

```
211.     return [x, y]
```

Finally, getPlayerMove() returns a two-item list with the XY coordinates of the player's valid move.

```
214. def getComputerMove(board, computerTile):
```

```
    # Given a board and the computer's tile, determine
where to
    # move and return that move as a [x, y] list.
    possibleMoves = getValidMoves(board, computerTile)
```

`getComputerMove()` and is where our Reversi AI is implemented. The `getValidMoves()` unction is very helpful for our AI. Normally we use the results from `getValidMoves()` for hints nove. Hints mode will print `'.'` period characters on the board to show the player all the potential noves they can make. But if we call `getValidMoves()` with the computer AI's tile (in `omputerTile`), we can get all the possible moves that the computer can make. We will select the best nove from this list.

The `random.shuffle()` Function

```
    # randomize the order of the possible moves
    random.shuffle(possibleMoves)
```

First, we are going to use the `random.shuffle()` function to randomize the order of moves in the `ossibleMoves` list. This is a function in the `random` module which will reorder the list that you pass o it. For example, try typing the following into the interactive shell:

```
import random
spam = [1, 2, 3, 4, 5, 6, 7, 8]
spam
random.shuffle(spam)
spam
random.shuffle(spam)
spam
random.shuffle(spam)
spam
```

```
>>> import random
>>> spam = [1, 2, 3, 4, 5, 6, 7, 8]
>>> spam
[1, 2, 3, 4, 5, 6, 7, 8]
>>> random.shuffle(spam)
>>> spam
[2, 8, 6, 5, 1, 4, 7, 3]
>>> random.shuffle(spam)
>>> spam
[7, 4, 2, 3, 8, 1, 5, 6]
>>> random.shuffle(spam)
>>> spam
[7, 6, 1, 4, 5, 3, 2, 8]
>>>
```

Your results may be different, because the reshuffling is random. As you can see, `random.shuffle()` itself does not have a return value. It modifies the list directly, much like our `resetBoard()` function does. This is why you must type `spam` into the shell to see the new value it has taken on.

Code Explanation continued...

We will explain why we want to shuffle the `possibleMoves` list, but first let's look at our algorithm.

```
222.     # always go for a corner if available.
223.     for x, y in possibleMoves:
224.         if isOnCorner(x, y):
225.             return [x, y]
```

First, we loop through every move in `possibleMoves` and if any of them are on the corner, we return that as our move. Corner moves are a good idea because once a tile has been placed on the corner, it can never be flipped over. Since `possibleMoves` is a list of two-item lists, we use the multiple assignment trick in our `for` loop to set `x` and `y`.

Because we immediately return on finding the first corner move in `random`, if `random` contains multiple corner moves we always go with the first one. But since `possibleMoves` was shuffled on line 220, it is completely random which corner move is first in the list.

```
227.     # Go through all the possible moves and remember
     the best scoring move
```

```
228.     bestScore = -1
229.     for x, y in possibleMoves:
230.         dupeBoard = getBoardCopy(board)
231.         makeMove(dupeBoard, computerTile, x, y)
232.         score = getScoreOfBoard(dupeBoard)
    [computerTile]
233.         if score > bestScore:
234.             bestMove = [x, y]
235.             bestScore = score
236.     return bestMove
```

If there are no corner moves, we will go through the entire list and find out which move gives us the highest score. The `for` loop will set `x` and `y` to every move in `possibleMoves`. `bestMove` will be set to the highest scoring move we've found so far, and `bestScore` will be set to its score. When the code in the loop finds a move that scores higher than `bestScore`, we will store that move and score as the new values of `bestMove` and `bestScore`.

In order to figure out the score of the possible move we are currently iterating on, we first make a duplicate game board data structure by calling `getBoardCopy()`. We want a copy so we can modify without changing the real game board data structure stored in the `board` variable.

Then we call `makeMove()`, passing the duplicate board instead of the real board. `makeMove()` will handle placing the computer's tile and the flipping the player's tiles on the duplicate board.

We call `getScoreOfBoard()` with the duplicate board, which returns a dictionary where the keys are `'X'` and `'O'`, and the values are the scores. `getScoreOfBoard()` does not know if the computer is `'X'` or `'O'`, which is why it returns a dictionary.

By making a duplicate board, we can simulate a future move and test the results of that move without changing the actual game board data structure. This is very helpful in deciding which move is the best possible move to make.

Pretend that `getScoreOfBoard()` returns the dictionary `{'X':22, 'O':8}` and `computerTile` is `'X'`. Then `getScoreOfBoard(dupeBoard)[computerTile]` would evaluate to `{'X':22, 'O':8}['X']`, which would then evaluate to `22`. If `22` is larger than `bestScore`, `bestScore` is set to `22` and `bestMove` is set to the current `x` and `y` values we are looking at. By the time this `for` loop is finished, we can be sure that `bestScore` is the highest possible score a move can make, and that move is stored in `bestMove`.

You may have noticed that on line 228 we first set `bestScore` to -1. This is so that the first move we look at in our `for` loop over `possibleMoves` will be set to the first `bestMove`. This will guarantee that `bestMove` is set to one of the moves when we return it.

Say that the highest scoring move in `possibleMoves` would give the computer a score of `42`. What

if there was more than one move in `possibleMoves` that would give this score? The `for` loop we use would always go with the first move that scored `42` points, because `bestMove` and `bestScore` only change if the move is *greater than* the highest score. A tie will not change `bestMove` and `bestScore`.

We do not always want to go with the first move in the `possibleMoves` list, because that would make our AI predictable by the player. But it is random, because on line 220 we shuffled the `possibleMoves` list. Even though our code always chooses the first of these tied moves, is random which of the moves will be first in the list because the order is random. This ensures that the AI will not be predictable when there is more than one best move.

```
239. def showPoints(playerTile, computerTile):
240.     # Prints out the current score.
241.     scores = getScoreOfBoard(mainBoard)
242.     print 'You have %s points. The computer has %s
     points.' % (scores[playerTile], scores[computerTile])
```

`showPoints()` simply calls the `getScoreOfBoard()` function and then prints out the player's score and the computer's score. Remember that `getScoreOfBoard()` returns a dictionary with the keys `'X'` and `'O'` and values of the scores for the X and O players.

That's all the functions we define for our Reversi game. The code starting on line 246 will implement the actual game and make calls to these functions when they are needed.

```
246. print 'Welcome to Reversi!'
247.
248. while True:
249.     # Reset the board and game.
250.     mainBoard = getNewBoard()
251.     resetBoard(mainBoard)
252.     playerTile, computerTile = enterPlayerTile()
253.     showHints = False
254.     turn = whoGoesFirst()
255.     print 'The ' + turn + ' will go first.'
```

The `while` loop on line 248 is the main game loop. The program will loop back to line 248 each time we want to start a new game. First we get a new game board data structure by calling `getNewBoard`

() and set the starting tiles by calling `resetBoard()`. `mainBoard` is the main game board data structure we will use for this program. The call to `enterPlayerTile()` will let the player type in whether they want to be `'X'` or `'O'`, which is then stored in `playerTile` and `computerTile`.

`showHints` is a boolean value that determines if hints mode is on or off. We originally set it to off by setting `showHints` to `False`.

The `turn` variable is a string will either have the string value `'player'` or `'computer'`, and will keep track of whose turn it is. We set `turn` to the return value of `whoGoesFirst()`, which randomly chooses who will go first. We then print out who goes first to the player on line 255.

```
257.     while True:
258.         if turn == 'player':
259.             # Player's turn.
260.             if showHints:
261.                 validMovesBoard =
     getBoardWithValidMoves(mainBoard, playerTile)
262.                 drawBoard(validMovesBoard)
263.             else:
264.                 drawBoard(mainBoard)
265.             showPoints(playerTile, computerTile)
```

The `while` loop that starts on line 257 will keep looping each time the player or computer takes a turn. We will break out of this loop when the current game is over.

Line 258 has an `if` statement whose body has the code that runs if it is the player's turn. (The else-block that starts on line 282 has the code for the computer's turn.) The first thing we want to do is display the board to the player. If hints mode is on (which it is if `showHints` is `True`), then we want to get a board data structure that has `'.'` period characters on every space the player could go.

Our `getBoardWithValidMoves()` function does that, all we have to do is pass the game board data structure and it will return a copy that also contains `'.'` period characters. We then pass this board to the `drawBoard()` function.

If hints mode is off, then we just pass `mainBoard` to `drawBoard()`.

After printing out the game board to the player, we also want to print out the current score by calling `showPoints()`.

```
266.             move = getPlayerMove(mainBoard,
```

```
    playerTile)
```

Next we let the player type in their move. `getPlayerMove()` handles this, and its return value is a two-item list of the X and Y coordinate of the player's move. `getPlayerMove()` makes sure that the move the player typed in is a valid move, so we don't have to worry about it here.

```
            if move == 'quit':
                print 'Thanks for playing!'
                sys.exit() # terminate the program
            elif move == 'hints':
                showHints = not showHints
                continue
            else:
                makeMove(mainBoard, playerTile, move
    [0], move[1])
```

If the player typed in the string `'quit'` for their move, then `getPlayerMove()` would have returned the string `'quit'`. In that case, we should call the `sys.exit()` to terminate the program.

If the player typed in the string `'hints'` for their move, then `getPlayerMove()` would have returned the string `'hints'`. In that case, we want to turn hints mode on (if it was off) or off (if it was on). The `showHints = not showHints` assignment statement handles both of these cases, because `not False` evaluates to `True` and `not True` evaluates to `False`. Then we run the `continue` statement to loop back (`turn` has not changed, so it will still be the player's turn after we loop).

Otherwise, if the player did not quit or toggle hints mode, then we will call `makeMove()` to make the player's move on the board.

```
            if getValidMoves(mainBoard, computerTile)
    == []:
                break
            else:
                turn = 'computer'
```

After making the player's move, we call `False` to see if the computer could possibly make any moves.

If False returns a blank list, then there are no more moves left that the computer could make (most likely because the board is full). In that case, we break out of the while loop and end the current game.

Otherwise, we set turn to 'computer'. The flow of execution skips the else-block and reaches the end of the while-block, so execution jumps back to the while statement on line 257. This time, however, it will be the computer's turn.

```
        else:
            # Computer's turn.
            drawBoard(mainBoard)
            showPoints(playerTile, computerTile)
            raw_input('Press Enter to see the computer\'s move.')
            x, y = getComputerMove(mainBoard, computerTile)
            makeMove(mainBoard, computerTile, x, y)
```

The first thing we do when it is the computer's turn is call drawBoard() to print out the board to the player. Why do we do this now? Because either the computer was selected to make the first move of the game, in which case we should display the original starting picture of the board to the player before the computer makes its move. Or the player has gone first, and we want to show what the board looks like after the player has moved but before the computer has gone.

After printing out the board with drawBoard(), we also want to print out the current score with a call to showPoints().

Next we have a call to raw_input() to pause the script while the player can look at the board. This is much like how we use raw_input() to pause the program in our Jokes chapter. Instead of using a print statement to print a string before a call to raw_input(), you can pass the string as a parameter to raw_input(). raw_input() has an optional string parameter. The string we pass in this call is 'Press Enter to see the computer\'s move.'.

After the player has looked at the board and pressed Enter (any text the player typed is ignored since we do not assign the return value of raw_input() to anything), we call getComputerMove() to get the X and Y coordinates of the computer's next move. We store these coordinates in variables x and y, respectively.

Finally, we pass x and y, along with the game board data structure and the computer's tile to the makeMove() function to change the game board to reflect the computer's move. Our call to getComputerMove() got the computer's move, and the call to makeMove() makes the move on the board.

```
289.             if getValidMoves(mainBoard, playerTile) == []:
290.                 break
291.             else:
292.                 turn = 'player'
```

Lines 289 to 292 are very similar to lines 276 to 279. After the computer has made its move, we check if there exist any possible moves the human player can make. If `getValidMoves()` returns an empty list, then there are no possible moves. That means the game is over, and we should break out of the `while` loop that we are in.

Otherwise, there is at least one possible move the player should make, so we should set `turn` to `'player'`. There is no more code in the while-block after line 292, so execution loops back to the `while` statement on line 257.

```
294.     # Display the final score.
295.     drawBoard(mainBoard)
296.     scores = getScoreOfBoard(mainBoard)
297.     print 'X scored %s points. O scored %s points.' % (scores['X'], scores['O'])
298.     if scores[playerTile] > scores[computerTile]:
299.         print 'You beat the computer by %s points! Congratulations!' % (scores[playerTile] - scores[computerTile])
300.     elif scores[playerTile] < scores[computerTile]:
301.         print 'You lost. The computer beat you by %s points.' % (scores[computerTile] - scores[playerTile])
302.     else:
303.         print 'The game was a tie!'
```

Line 294 is the first line beyond the while-block that started on line 257. This code is executed when we have broken out of that `while` loop, either on line 290 or 277. (The `while` statement's condition on line 257 is simply the value `True`, so we can only exit the loop through `break` statements.)

At this point, the game is over. We should print out the board and scores, and determine who won the game. `getScoreOfBoard()` will return a dictionary with keys `'X'` and `'O'` and values of both players' scores. By checking if the player's score is greater than, less than, or equal to the computer's

score, we can know if the player won, if the player lost, or if the player and computer tied.

Subtracting one score from the other is an easy way to see by how much one player won over the other. Our `print` statements on lines 29 and 301 use string interpolation to put the result of this subtraction into the string that is printed.

```
    if not playAgain():
        break

```

The game is now over and the winner has been declared. We should call our `playAgain()` function, which returns `True` if the player typed in that they want to play another game. If `playAgain()` returns `False` (which makes the `if` statement's condition `True`), we break out of the `while` loop (the one that started on line 248), and since there are no more lines of code after this while-block, the program terminates.

Otherwise, `playAgain()` has returned `True` (which makes the `if` statement's condition `False`), and so execution loops back to the `while` statement on line 248 and a new game board is created.

Tips for Inventing Your Own Games

That does it for this book as far as games go. The next chapter expands on creating new Reversi AIs, and having AIs play against each other instead of against a human player. Using the programming techniques in this book, you can start building your own simple games. Here are a few pointers:

- Drawing out a flow chart before you start writing code might help you remember everything that you want to happen in your game.
- Use the interactive shell to test out what an expression or function call might evaluate to. The shell is a great way to experiment with different functions.
- If something strange is happening when you run your program, try adding some `print` statements in the middle of the code to print out the values of different variables. Or you can use `print` statements to check if some functions are being called when you expect them too, or how often a loop is iterated.
- If you ever find yourself writing identical code in several places of the program, see if you can put that code in a function and call that function several times. This way, if you want to change the code in the function, there is only one place you have to make changes.

Things Covered In This Chapter:

- The `bool()` Function

- The `random.shuffle()` Function

Chapter 11 - AI Simulation

"Computer vs. Computer" Games

The Reversi AI algorithm was very simple, but it beats me almost every time I play it. This is because the computer can process instructions very fast, so checking each possible position on the board and selecting the highest scoring move is easy. If I took the time to look at every space on the board and write down the score of each possible move, it would take a long time for me to find the best move.

Did you notice that our Reversi program in Chapter 11 had two functions, `getPlayerMove()` and `getComputerMove()`, which both returned the move selected as a two-item list like `[x, y]`? The both also had the same parameters, the game board data structure and which tile they were. `getPlayerMove()` decided which `[x, y]` move to return by letting the player type in the coordinates. `getComputerMove()` decided which `[x, y]` move to return by running the Reversi AI algorithm.

What happens when we replace the call to `getPlayerMove()` with a call to `getComputerMove()`? Then the player never types in a move, it is decided for them! The computer is playing against itself!

Save the old reversi.py file as AISim1.py by clicking on File and then Save As, and then entering AISim1.py for the file name and clicking Ok. This will create a copy of our Reversi source code as a new file that we can make changes to, while leaving the original Reversi game the same (we may want to play it again). Change the following code in AISim1.py:

```
266.             move = getPlayerMove(mainBoard,
    playerTile)
```

To this (the change is in bold):

```
266.             move = getComputerMove(mainBoard,
    playerTile)
```

And run the program. Notice that the game still asks you if you want to be X or O, but it will not ask you to enter in any moves. When we replaced `getPlayerMove()`, we no longer call any code that takes this input from the player. We still press Enter after the original computer's moves (because of the `raw_input('Press Enter to see the computer\'s move.')` on line 285), but the game plays itself!

Let's make some other changes. All of the functions we defined for Reversi can stay the same. But change the entire main section of the program (line 246 and on) to look like the following:

AISim1.py

```
246. print 'Welcome to Reversi!'
247.
248. while True:
249.     # Reset the board and game.
250.     mainBoard = getNewBoard()
251.     resetBoard(mainBoard)
252.     if whoGoesFirst() == 'player':
253.         turn = 'X'
254.     else:
255.         turn = 'O'
256.     print 'The ' + turn + ' will go first.'
257.
258.     while True:
259.         drawBoard(mainBoard)
260.         scores = getScoreOfBoard(mainBoard)
261.         print 'X has %s points. O has %s points' % (scores['X'], scores['O'])
262.         raw_input('Press Enter to continue.')
263.
264.         if turn == 'X':
265.             # X's turn.
266.             otherTile = 'O'
267.             x, y = getComputerMove(mainBoard, 'X')
268.             makeMove(mainBoard, 'X', x, y)
269.         else:
270.             # O's turn.
271.             otherTile = 'X'
272.             x, y = getComputerMove(mainBoard, 'O')
273.             makeMove(mainBoard, 'O', x, y)
274.
275.         if getValidMoves(mainBoard, otherTile) == []:
276.             break
277.         else:
278.             turn = otherTile
279.
280.     # Display the final score.
281.     drawBoard(mainBoard)
282.     scores = getScoreOfBoard(mainBoard)
283.     print 'X scored %s points. O scored %s points.' % (scores['X'], scores['O'])
284.
```

```
    if not playAgain():
        sys.exit()
```

Code Explanation

When you run the AISim1.py program, all you can do is press Enter for each turn until the game ends. Run through a few games and watch the computer play itself. Since both the X and O players are using the same algorithm, it really is just a matter of luck to see who wins. The X player will win half the time, and the O player will win half the time.

But what if we created a new algorithm? Then we could set this new AI against the one implemented in `getComputerMove()`, and see which one is better. Let's make some changes to our program. Click on File and then Save As, and save this file as AISim2.py so that we can make changes without affecting AISim1.py.

Add the following code. The additions are in bold, and some lines have been removed:

AISim2.py

```
print 'Welcome to Reversi!'

xwins = 0
owins = 0
ties = 0
numGames = int(raw_input('Enter number of games to run: '))

for game in range(numGames):
    print 'Game #%s:' % (game),
    # Reset the board and game.
    mainBoard = getNewBoard()
    resetBoard(mainBoard)
    if whoGoesFirst() == 'player':
        turn = 'X'
    else:
        turn = 'O'

    while True:
        if turn == 'X':
            # X's turn.
            otherTile = 'O'
            x, y = getComputerMove(mainBoard, 'X')
            makeMove(mainBoard, 'X', x, y)
        else:
```

```
270.             # O's turn.
271.             otherTile = 'X'
272.             x, y = getComputerMove(mainBoard, 'O')
273.             makeMove(mainBoard, 'O', x, y)
274.
275.         if getValidMoves(mainBoard, otherTile) == []:
276.             break
277.         else:
278.             turn = otherTile
279.
280.     # Display the final score.
281.     scores = getScoreOfBoard(mainBoard)
282.     print 'X scored %s points. O scored %s points.' %
    (scores['X'], scores['O'])
283.
284.     if scores['X'] > scores['O']:
285.         xwins += 1
286.     elif scores['X'] < scores['O']:
287.         owins += 1
288.     else:
289.         ties += 1
290.
291. numGames = float(numGames)
292. xpercent = round(((xwins / numGames) * 100), 2)
293. opercent = round(((owins / numGames) * 100), 2)
294. tiepercent = round(((ties / numGames) * 100), 2)
295. print 'X wins %s games (%s%%), O wins %s games (%s%%),
    ties for %s games (%s%%) of %s games total.' % (xwins,
    xpercent, owins, opercent, ties, tiepercent, numGames)
```

Code Explanation

We have added the variables `xwins`, `owins`, and `ties` to keep track of how many times X wins, O wins, and when they tie. Lines 284 to 289 increment these variables at the end of each game, before it loops back to start a brand new game.

We have removed most of the `print` statements from the program, and the calls to `drawBoard()`. When you run AISim2.py, it asks you how many games you wish to run. Now that we've taken out the call to `drawBoard()` and replace the `while True:` loop with a `for game in range (numGames):` loop, we can run a number of games without stopping for the user to type anything. Here is a sample run where we run ten games of computer vs. computer Reversi:

Sample Run

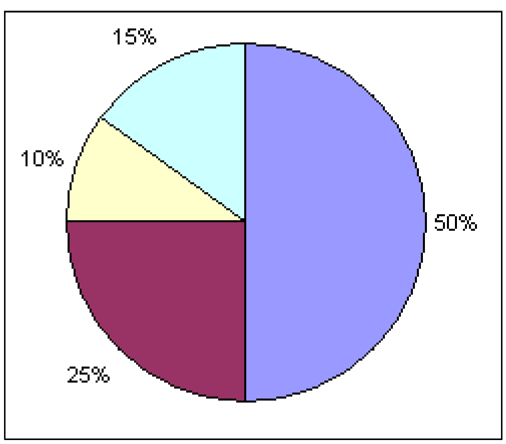
```
Welcome to Reversi!
Enter number of games to run: 10
Game #0: X scored 40 points. O scored 23 points.
Game #1: X scored 24 points. O scored 39 points.
Game #2: X scored 31 points. O scored 30 points.
Game #3: X scored 41 points. O scored 23 points.
Game #4: X scored 30 points. O scored 34 points.
Game #5: X scored 37 points. O scored 27 points.
Game #6: X scored 29 points. O scored 33 points.
Game #7: X scored 31 points. O scored 33 points.
Game #8: X scored 32 points. O scored 32 points.
Game #9: X scored 41 points. O scored 22 points.
X wins 5 games (50.0%), O wins 4 games (40.0%), ties for 1
games (10.0%) of 10.0 games total.
```

Because the algorithm does have a random part, your run might not have the exact same numbers as above.

Printing things out to the screen slows the computer down, but now that we have removed that code, the computer can run an entire game of Reversi in about a second or two. Think about it. Each time our program printed out one of those lines, it ran through an entire game (which is about fifty or sixty moves, each move carefully checked to be the one that gets the most points).

Percentages

Percentages are a portion of a total amount, and range from 0% to 100%. If you had 100% of a pie, you would have the entire pie. If you had 0% of a pie, you wouldn't have any pie at all. 50% of the pie would be half of the pie. A pie is a common image to use for percentages. In fact, there is a kind of chart called a **pie chart** which shows how much of the full total a certain portion is. Here is a pie chart with 10%, 15%, 25%, and 50% portions:



We can calculate the percentage with division. To get a percentage, divide the part you have by the total, and then multiply by one hundred. For example, if X won 50 out of 100 games, you would

calculate the expression `50 / 100.0`, which would evaluate to `0.5`. We multiple this by `100` to get a percentage (in this case, 50%).

Integer Division

Did you notice that we divided `50 / 100.0`, not `50 / 100`? The reason behind this is that there are two types of division in the Python language. Regular division is done when at least one of the numbers in the division expression is a float data type, that is, a number that has a decimal point and then a certain fraction after it. Regular division will evaluate to another float value. (For example, `50 / 100.0` evaluates to `0.5`.)

However, if both numbers in the division expression are integers (that is, whole numbers without a decimal point), then Python will do integer division. **Integer division** is division of two integer numbers that evaluate to a rounded-down integer.

For example, the expression `20 / 3.0` or the expression `20.0 / 3` will evaluate to `6.666666666666667`. However, the expression `20 / 3` evaluates to the integer `6`. This is because twenty divided by three is six, with a remainder of two. In integer division, the remainder part is dropped.

We want to use regular division when we calculate our percentages, because otherwise instead of a float value like `0.5`, integer division for percentages will always evaluate to `0`.

The `round()` Function

The `round()` function will round a float number to the nearest whole float number. Try typing the following into the interactive shell:

```
round(10.0)
round(10.2)
round(8.7)
round(4.5)
round(3.5)
round(3.4999)
round(2.5422, 2)
```

```
>>> round(10.0)
10.0
>>> round(10.2)
10.0
>>> round(8.7)
9.0
>>> round(4.5)
5.0
>>> round(3.5)
4.0
>>> round(3.4999)
3.0
>>> round(2.5422, 2)
2.54
>>>
```

As you can see, whenever the fraction part of a number is `.5` or greater, the number is rounded up. Otherwise, the number is rounded down. The `round()` function also has an optional parameter, where you can specify to what place you wish to round the number to. For example, the expression `round(2.5422, 2)` evaluates to `2.54`.

Code Explanation continued...

```
numGames = float(numGames)
xpercent = round(((xwins / numGames) * 100), 2)
opercent = round(((owins / numGames) * 100), 2)
tiepercent = round(((ties / numGames) * 100), 2)
print 'X wins %s games (%s%%), O wins %s games (%s%%),
ties for %s games (%s%%) of %s games total.' % (xwins,
xpercent, owins, opercent, ties, tiepercent, numGames)
```

The code at the bottom of our program will show the user how many wins X and O had, how many ties there were, and how what percentages these make up. Statistically, the more games you run, the more accurate your percentages will be. If you only ran ten games, and X won three of them, then it would seem that X's algorithm only wins 30% of the time. However, if you run a hundred, or even a thousand games, then you may find that X's algorithm wins closer to 50% (that is, half) of the games.

To find the percentages, we divide the number of wins or ties by the total number of games. We convert `numGames` to a float to ensure we do not use integer division in our calculation. Then we multiple the result by `100`. However, we may end up with a number like `66.66666666666667`. So we pass this number to the `round()` function with the second parameter of `2)`, so it will return a float like `66.67` instead (which is much more readable).

Let's try another experiment. Run AISim2.py again, but this time have it run a hundred games:

Sample Run

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 42 points. O scored 18 points.
Game #1: X scored 26 points. O scored 37 points.
Game #2: X scored 34 points. O scored 29 points.
Game #3: X scored 40 points. O scored 24 points.

...skipped for brevity...

Game #96: X scored 22 points. O scored 39 points.
Game #97: X scored 38 points. O scored 26 points.
Game #98: X scored 35 points. O scored 28 points.
Game #99: X scored 24 points. O scored 40 points.
X wins 46 games (46.0%), O wins 52 games (52.0%), ties for 2
games (2.0%) of 100.0 games total.
```

Depending on how fast your computer is, this run might have taken a about a couple minutes. We can see that the results of all one hundred games still evens out to about fifty-fifty, because both X and O are using the same algorithm to win.

Let's add some new functions with new algorithms. But first click on File, then Save As, and save this file as AISim3.py. Before the `print 'Welcome to Reversi!'` line, add these functions:

AISim3.py

```
def getRandomMove(board, tile):
    # Return a random move.
    return random.choice( getValidMoves(board, tile) )


def isOnSide(x, y):
    return x == 0 or x == 7 or y == 0 or y ==7


def getCornerSideBestMove(board, tile):
    # Return a corner move, or a side move, or the best move.
    possibleMoves = getValidMoves(board, tile)

    # randomize the order of the possible moves
    random.shuffle(possibleMoves)

    # always go for a corner if available.
```

```
    for x, y in possibleMoves:
        if isOnCorner(x, y):
            return [x, y]

    # if there is no corner, return a side move.
    for x, y in possibleMoves:
        if isOnSide(x, y):
            return [x, y]

    return getComputerMove(board, tile)


def getSideBestMove(board, tile):
    # Return a corner move, or a side move, or the best move.
    possibleMoves = getValidMoves(board, tile)

    # randomize the order of the possible moves
    random.shuffle(possibleMoves)

    # return a side move, if available
    for x, y in possibleMoves:
        if isOnSide(x, y):
            return [x, y]

    return getComputerMove(board, tile)


def getWorstMove(board, tile):
    # Return the move that flips the least number of tiles.
    possibleMoves = getValidMoves(board, tile)

    # randomize the order of the possible moves
    random.shuffle(possibleMoves)

    # Go through all the possible moves and remember the best scoring move
    worstScore = 64
    for x, y in possibleMoves:
        dupeBoard = getBoardCopy(board)
        makeMove(dupeBoard, tile, x, y)
        score = getScoreOfBoard(dupeBoard)[tile]
        if score < worstScore:
            worstMove = [x, y]
            worstScore = score

```

```
306.     return worstMove
307.
308.
309. def getCornerWorstMove(board, tile):
310.     # Return a corner, a space, or the move that flips the least number of tiles.
311.     possibleMoves = getValidMoves(board, tile)
312.
313.     # randomize the order of the possible moves
314.     random.shuffle(possibleMoves)
315.
316.     # always go for a corner if available.
317.     for x, y in possibleMoves:
318.         if isOnCorner(x, y):
319.             return [x, y]
320.
321.     return getWorstMove(board, tile)
322.
323.
324.
325. print 'Welcome to Reversi!'
```

Code Explanation

A lot of these functions are very similar to one another, and some of them use the new `isOnSide()` function. Here's a review of the new algorithms we've made:

`getRandomMove()`	Randomly choose a valid move to make.
`getCornerSideBestMove()`	Take a corner move if available. If there is no corner, take a space on the side. If no sides are available, use the regular `getComputerMove()` algorithm.
`getSideBestMove()`	Take a side space if there is one available. If not, then use the regular `getComputerMove()` algorithm (side spaces are chosen before corner spaces).
`getWorstMove()`	Take the space that will result in the *fewest* tiles being flipped.
`getCornerWorstMove()`	Take a corner space, if available. If not, use the `getWorstMove()` algorithm.

Now the only thing to do is replace one of the `getComputerMove()` calls in the main part of the program with one of the new functions. Then we can run several games and see how often one algorithm wins over the other. First, let's replace O's algorithm with the one in `getComputerMove()` with `getRandomMove()` on line 386:

```
386.            x, y = getRandomMove(mainBoard, 'O')
```

When we run the program with a hundred games now, it may look something like this:

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 25 points. O scored 38 points.
Game #1: X scored 32 points. O scored 32 points.
Game #2: X scored 15 points. O scored 0 points.
Game #3: X scored 50 points. O scored 14 points.

...skipped for brevity...

Game #96: X scored 31 points. O scored 33 points.
Game #97: X scored 41 points. O scored 23 points.
Game #98: X scored 33 points. O scored 31 points.
Game #99: X scored 45 points. O scored 19 points.
X wins 84 games (84.0%), O wins 15 games (15.0%), ties for 1
games (1.0%) of 100.0 games total.
```

Wow! X win far more often than O did. That means that the algorithm in `getComputerMove()` (take any available corners, otherwise take the space that flips the most tiles) wins more games than the algorithm in `getRandomMove()` (which just makes moves randomly). This makes sense, because making intelligent choices is usually going to be better than just choosing things at random.

What if we changed X's algorithm to also use the algorithm in `getRandomMove()`? Let's find out by changing X's function call from `getComputerMove()` to `getRandomMove()` and running the program again.

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 37 points. O scored 24 points.
Game #1: X scored 19 points. O scored 45 points.

...skipped for brevity...

Game #98: X scored 27 points. O scored 37 points.
Game #99: X scored 38 points. O scored 22 points.
X wins 42 games (42.0%), O wins 54 games (54.0%), ties for 4
games (4.0%) of 100.0 games total.
```

As you can see, when both players are making random moves, they each win about 50% of the time.

(In the above case, O just happen to get lucky and won a little bit more than half of the time.)

Just like moving on the corner spaces is a good idea because they cannot be flipped, moving on the side pieces may also be a good idea. On the side, the tile has the edge of the board and is not as out in the open as the other pieces. The corners are still preferable to the side spaces, but moving on the sides (even when there is a move that can flip more pieces) may be a good strategy.

Change X's algorithm to use `getComputerMove()` (our original algorithm) and O's algorithm to use `getCornerSideBestMove()`, and let's run a hundred games to see which is better. Try changing the function calls and running the program again.

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 52 points. O scored 12 points.
Game #1: X scored 10 points. O scored 54 points.

...skipped for brevity...

Game #98: X scored 41 points. O scored 23 points.
Game #99: X scored 46 points. O scored 13 points.
X wins 65 games (65.0%), O wins 31 games (31.0%), ties for 4
games (4.0%) of 100.0 games total.
```

Wow! That's unexpected. It seems that choosing the side spaces over a space that flips more tiles is a bad strategy to use. The benefit of the side space is not greater than the cost of choosing a space that flips fewer of the opponent's tiles. Can we be sure of these results? Let's run the program again, but this time let's have the program play one thousand games. This may take a few minutes for your computer to run (but it would take days for you to do this by hand!) Try changing the function calls and running the program again.

```
Welcome to Reversi!
Enter number of games to run: 1000
Game #0: X scored 20 points. O scored 44 points.
Game #1: X scored 54 points. O scored 9 points.

...skipped for brevity...

Game #998: X scored 38 points. O scored 23 points.
Game #999: X scored 38 points. O scored 26 points.
X wins 611 games (61.1%), O wins 363 games (36.3%), ties for
26 games (2.6%) of 1000.0 games total.
```

The more accurate statistics from the thousand-games run are about the same as the statistics from the hundred-games run. It seems that choosing the move that flips the most tiles is a better idea than choosing a side move.

Now set the X player's algorithm to use `getComputerMove()` and the O player's algorithm to `getWorstMove()`, and run a hundred games. Try changing the function calls and running the program again.

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 50 points. O scored 14 points.
Game #1: X scored 38 points. O scored 8 points.

...skipped for brevity...

Game #98: X scored 36 points. O scored 16 points.
Game #99: X scored 19 points. O scored 0 points.
X wins 98 games (98.0%), O wins 2 games (2.0%), ties for 0
games (0.0%) of 100.0 games total.
```

Whoa! The algorithm in `getWorstMove()`, which always choose the move that flips the *fewest* tiles, will almost always lose to our regular algorithm. This isn't really surprising at all. How about when we replace `getWorstMove()` with `getCornerWorstMove()`, which is the same algorithm except it takes any available corner pieces. Try changing the function calls and running the program again.

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 36 points. O scored 7 points.
Game #1: X scored 44 points. O scored 19 points.

...skipped for brevity...

Game #98: X scored 47 points. O scored 17 points.
Game #99: X scored 36 points. O scored 18 points.
X wins 94 games (94.0%), O wins 6 games (6.0%), ties for 0
games (0.0%) of 100.0 games total.
```

The `getCornerWorstMove()` still loses most of the games, but it seems to win a few more games than `getWorstMove()` (6% compared to 2%). Does taking the corner spaces when they are available really make a difference? We can check by setting X's algorithm to `getWorstMove()` and O's algorithm to `getCornerWorstMove()`, and then running the program. Try changing the function calls and running the program again.

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 25 points. O scored 39 points.
Game #1: X scored 26 points. O scored 33 points.

...skipped for brevity...
```

```
Game #98: X scored 36 points. O scored 25 points.
Game #99: X scored 29 points. O scored 35 points.
X wins 32 games (32.0%), O wins 67 games (67.0%), ties for 1
games (1.0%) of 100.0 games total.
```

Yes, it does seem like taking the algorithm that takes the corners when it can does translate into more wins. While we have found out that going for the sides makes you lose more often, going for the corners is always a good idea.

Learning New Things by Running Simulation Experiments

This chapter didn't really cover a game, but it modeled various strategies for Reversi. If we thought that taking side moves in Reversi was a good idea, we would have to spend days, even weeks, carefully playing games of Reversi by hand and writing down the results. But if we know how to program a computer to play Reversi, then we can have the computer play Reversi using these strategies for us. If you think about it, you will realize that the computer is executing millions of lines of our Python program in seconds! Your experiments with the simulation of Reversi can help you learn more about playing Reversi in real life.

And it is all because you know exactly how to instruct the computer to do it, step by step, line by line. You can speak the computer's language, and get it to do large amounts of data processing and number crunching for you. This is a very useful skill, and I hope you will continue to learn more about Python programming. (And there is still more to learn!)

The next step you can take is looking at the help file that comes with Python. You can access this by clicking on the Start button in Windows' lower left corner, then going to Programs (or All Programs), then the Python 2.5 folder, and then clicking the "Python Manuals" link.



Another way you can find out more about Python is searching the Internet. Go to the website http://google.com and search for "Python programming" or "Python tutorials" to find web sites that can teach you more about Python programming.

Now get going and invent your own games. And good luck!

Things Covered In This Chapter:

- Simulations
- Percentages
- Pie Charts
- Integer Division
- The `round()` Function

Glossary

absolute value - The positive form of a negative number. For example, the absolute value of -2 is 2. The absolute value of a positive number is simply the positive number itself.

AI - see, artificial intelligence

algorithm - A series of instructions to compute something.

applications - A program that is run by an operating system. See also, program.

arguments - The values that are passed for parameters in a function call.

artificial intelligence - Code or a program that can intelligent make decisions (for example, decisions when playing a game) in response to user actions.

ASCII art - Using text characters and spaces to draw simple pictures.

assembly language - The simplest programming language. Assembly language instructions are a human-readable form that can directly translate into machine code instructions.

assignment operator - The = sign. Used to assign values to variables.

assignment statement - A line of code that assigns a value to a variable using the assignment operator. This defines, that is, creates the variable when used with a new variable. For example: `spam = 42`

asterisk - The * symbol. The asterisk is used as a multiplication sign.

augmented assignment operator - The `+=`, `-=`, `*=`, and `/=` operators. The assignment `spam += 42` is equivalent to `spam = spam + 42`.

block - A group of lines of code with the same amount of indentation. Blocks can contain other blocks of greater indentation inside them.

boolean - A data type with only two values, `True` and `False`.

boolean operator - Boolean operators include `and`, `or`, and `not`.

break statement - The `break` statement immediately jumps out of the current `while` or `for` loop to the first line after the end of the loop's block.

brute force - In cryptography, to try every possible key in order to decrypt an encrypted message.

caesar cipher - A simple substitution cipher in which each symbol is replaced by one and only one other symbol.

cartesian coordinate system - A system of coordinates used to identify exact points in some area of space (such as the monitor, or on a game board). Cartesian coordinates systems commonly have two coordinates, one of the X-axis (that is, the horizontal left-right axis) and one of the Y-axis (that is, the vertical up-down axis).

case-sensitivity - Declaring different capitalizations of a name to mean different things. Python is a case-sensitive language, so `spam`, `Spam`, and `SPAM` are three different variables.

central processing unit - CPU, the main chip that your computer uses to process software instructions.

cipher - In cryptography, an algorithm used to encrypt and decrypt messages with a certain key.

ciphertext - In cryptography, the encrypted form of a message.

comment - Part of the source code that is ignored by the Python interpreter. Comments are there to remind the programmer about something about the code. Comments begin with a # sign and go on for the rest of the line.

commutative property - The property of addition and multiplication that describes how the order of the numbers being added or multiplied does not matter. For example, 2 + 4 = 6, and 4 + 2 = 6. Also, 3 * 5 = 15, and 5 * 3 = 15.

comparison operators - The operators `<` ("less than"), `<=` ("less than or equal to"), `>` ("greater than"), `>=` ("greater than or equal to"), `==` ("equal to"), and `!=` ("not equal too").

condition - Another name for an expression, one that exists in an `if` or `while` statement that evaluates to a boolean `True` or `False` value.

constant variables - Variables whose values do not change. Constant variables are often used because it is easier to type the name of the variable then the value that they store. As a convention, constant variable names are typed in all uppercase letters.

convention - A way of doing things that is not required, but is usually done to make a task easier.

conversion specifiers - The text inside a string that makes use of string interpolation. The most common conversion specifier is `%s`, which specifies that the variable it interpolates should be converted to a string.

cpu - see, Central Processing Unit

cryptanalysis - The science of breaking secret codes and ciphers.

cryptography - The science of making secret codes and ciphers.

data types - A category of values. Some types in Python are: strings, integers, floats, boolean, lists, and NoneType.

decrementing - To decrease a numeric value by one.

decrypting - To convert an encrypted message to the readable plaintext version.

def statement - A statement that defines a new function. The def statement begins with the `def` keyword, followed by the function name and a set of parentheses, with any number of parameter names delimited by commas. At the end is a : colon character. For example, `def funcName(param1, param2):`

delimit - To separate with. For example, the string `'cats,dogs,mice'` is delimited with commas.

dictionary - A container data type that can store other values. Values are accessed by a key. For example, `spam['foo'] = 42` assigns the key `'foo'` of the `spam` dictionary the value `42`.

else statement - An `else` statement always follows an `if` statement, and the code inside the else-block is executed if the `if` statement's condition was `False`.

empty list - The list `[]`, which contains no values and has a length of zero. See also, empty string.

empty string - The string `''`, which contains no characters and has a length of zero. See also, empty list.

encrypting - To convert a message into a form that resembles garbage data, and cannot be understood except by someone who knows the ciphr and key used to encrypt the message.

escape character - Escape characters allow the programmer to specify characters in Python that are difficult or impossible to type into the source code. All escape characters are preceeded by a \ forward backslash character. For example, `\n` displays a newline character when it is printed.

evaluate - Reducing an expression down to a single value. The expression `2 + 3 + 1` evaluates to the value `6`.

execute - The Python interpreter executes lines of code, by evaluating any expressions or performing the task that the code does.

exit - When a program ends. "Terminate" means the same thing.

expression - Values and function calls connected by operators. Expressions can be evaluated down to a single value.

file editor - A program used to type in or change files, including files of Python source code. The IDLE program has a file editor that you use to type in your programs.

floating point numbers - Numbers with fractions or decimal points are not integers. The numbers `3.5` and `42.1` and `5.0` are floating point numbers.

flow chart - A chart that informally shows the flow of execution for a program, and the main events that occur in the program and in what order.

flow control statements - Statements that cause the flow of execution to change, often depending on conditions. For example, a function call sends the execution to the beginning of a function. Also, a loop causes the execution to iterate over a section of code several times.

flow of execution - The order that Python instructions are executed. Usually the Python interpreter will start at the top of a program and go down executing one line at a time. Flow control statements can move the flow of execution to different parts of code in the program.

function - A collection of instructions to be executed when the function is called. Functions also have a return value, which is the value that a function call evaluates to.

function call - A command to pass execution to the code contained inside a function, also passing arguments to the function. Function calls evaluate to the return value of the function.

garbage data - Random data or values that have no meaning.

global scope - The scope of variables outside of all functions. Python code in the global scope cannot see variables inside any function's local scope.

hard-coding - Using a value in a program, instead of using a variable. While a variable could allow the program to change, by hard-coding a value in a program, the value stays permanently fixed unless the source code is changed.

hardware - The parts of a computer that you can touch, such as the keyboard, monitor, case, or mouse. See also, software.

higher-level programming languages - Programming languages that humans can understand, such as Python. An interpreter can translate a higher-level language into machine code, which is the only language computers can understand.

IDLE - Interactive DeveLopment Environment. IDLE is a program that helps you type in your programs and games.

I/O - Input/Output. This is a term used in reference of the data that is sent into a program (input) and that is produced by the program (output).

immutable sequence - A container data type that cannot have values added or deleted from it. In Python, the two immutable sequence data types are strings and tuples.

import statement - A line of code with the `import` keyword followed by the name of a module. This allows you to call any functions that are contained in the module.

incrementing - To increase the value of a numeric variable by one.

indentation - The indentation of a line of code is the number of spaces before the start of the actual code. Indentation in Python is used to mark when blocks begin and end. Indentation is usually done in multiples of four spaces.

index - An integer between square brackets that is placed at the end of an ordered container variable (most often a list) to evaluate to a specific item in that container. The first index starts at 0, not 1. For example, if `spam` refers to the list `['a', 'b', 'c', 'd']`, then `spam[2]` evaluates to `'c'`.

index error - An index error occurs when you attempt to access an index that does not exist. This is much like using a variable that does not exist. For example, if `spam` refers to the list `['a', 'b', 'c', 'd']`, then `spam[10]` would cause an index error.

input - The text or data that the user or player enters into a program, mostly from the keyboard.

integer division - Division that ignores any remainder and rounds the evaluated number down. Integer division occurs when both numbers in the division expression are integers. For example, `20 / 7` evaluates to the integer `6`, even though the answer is 6.666 or 6 remainder 2.

integers - Integers are whole numbers like `4` and `99` and `0`. The numbers `3.5` and `42.1` and `5.0` are not integers.

interactive shell - A part of IDLE that lets you execute Python code one line at a time. It allows you to immediately see what value the expression you type in evaluates to.

interpreter - A program that translates instructions written in a higher-level programming language (such as Python) to machine code that the computer can understand and execute.

iteration - A single run through of the code in a loop's block. For example, if the code in a while-block is executed ten times before execution leaves the loop, we say that there were ten iterations of the while-block's code.

key-value pairs - In dictionary data types, keys are values that are used to access the values in a dictionary, much like a list's index is used to access the values in a list. Unlike lists, dictionary keys can be of any data type, not just integers.

keys - In dictionaries, keys are the indexes used to

keys - In cryptography, a specific value (usuaully a number) that determines how a cipher encrypts a message. To decrypt the message, you must know both the cipher and the key value that was used.

list - The main container data type, lists can contain several other values, including other lists. Values in lists are accessed by an integer index between square brackets. For example, if `spam` is assigned the list `['a', 'b', 'c']`, then `spam[2]` would evaluate to `'c'`.

list concatenation - Combining the contents of one list to the end of another with the + operator. For example, `[1, 2, 3] + ['a', 'b', 'c']` evaluates to `[1, 2, 3, 'a', 'b', 'c']`.

local scope - The scope of variables inside a single functions. Python code inside a function can read the value of variables in the global scope, but any changes or new variables made will only exist while execution is inside that function call.

loop - A block of code inside a loop (after a `for` or `while` statement) will repeatedly execute until some condition is met.

loop unrolling - Replacing code inside a loop with multiple copies of that code. For example, instead of `for i in range(10): print 'Hello'`, you could unroll that loop by having ten lines of `print 'Hello'`

machine code - The language that the computer's CPU understands. Machine code instructions are series of ones and zeros, and is generally unreadable by humans. Interpreters (such as the Python interpreter) translate a higher-level language into machine code.

methods - Functions that are associated with values of a data type. For example, the string method `upper()` would be invoked on a string like this: `'Hello'.upper()`

module - A separate Python program that can be included in your programs so that you can make use of the functions in the module.

modulus operator - The "remainder" operator that is represented with a `%` percent sign. For example, while 20 / 7 is 6 with a remainder of 2, `20 % 7` would evaluate to `2`.

mutable sequence - A container data type that is ordered and can have values added or removed from it. Lists are a mutable sequence data type in Python.

negative numbers - All numbers less than 0. Negative numbers have a minus sign in front of them to differentiate them from positive numbers, for example, -42 or -10.

nested loops - Loops that exist inside other loops.

None - The only value in the NoneType data type. "None" is often used to represent the lack of a value.

operating system - A large program that runs other software programs (called applications) the same way on different hardware. Windows, Mac OS, and Linux are examples of operating systems.

operators - Operators connect values in expressions. Operators include `+`, `-`, `*`, `/`, `and`, and `or`

ordinal - In ASCII, the number that is represented by an ASCII character. For example, the ASCII character `'A'` has the ordinal 65.

origin - In cartesian coordinate systems, the point at the coordinates 0, 0.

OS - see, operating system

output - The text that a program produces for the user. For example, `print` statements produce output.

overwrite - To replace a value stored in a variable with a new value.

parameter - A variable that is specified to have a value passed in a function call. For example, the statement `def spam(eggs, cheese)` defines a function with two parameters named `eggs` and `cheese`.

pie chart - A circular chart that shows percentage portions as portions of the entire circle.

plaintext - The decrypted, human-readable form of a message.

player - A person who plays the computer game.

positive numbers - All numbers equal to or greater than 0.

pound sign - The # sign. Pound signs are used to begin comments.

print statement - The `print` keyword followed by a value that is to be displayed on the screen.

program - A collection of instructions that can process input and produce output when run by computer.

programmer - A person who writes computer programs.

reference - Rather than containing the values themselves, list variables actually contain references to lists. For example, `spam = [1, 2, 3]` assigns `spam` a reference to the list. `cheese = spam` would copy the reference to the list `spam` refers to. Any changes made to the `cheese` or `spam` variable would be reflected in the other variable.

return statement - The `return` followed by a single value, which is what the call to the function the return statement is in will evaluate to.

return value - The value that a call to the function will evaluate to. You can specify what the return value is with the `return` keyword followed by the value. Functions with no return statement will return the value `None`.

scope - See, local scope and global scope.

sequence - A sequence data type is an ordered container data type, and have a "first" or "last" item. The sequence data types in Python are lists, tuples, and strings. Dictionaries are not sequences, they are unordered. See also, unordered.

shell - see, interactive shell

simple substitution ciphers -

slice - A subset of values in a list. These are accessed using the : colon character in between the square brackets. For example, if `spam` has the value `['a', 'b', 'c', 'd', 'e', 'f']`, then the slice `spam[2:4]` has the value `['c', 'd']`. Similar to a substring.

software - see, program

source code - The text that you type in to write a program.

statement - A command or line of Python code that does not evaluate to a value.

string concatenation - Combining two strings together with the + operator to form a new string. For example, `'Hello ' + 'World!'` evaluates to the string `'Hello World!'`

string formatting - Another term for string interpolation.

string interpolation - Using conversion specifiers in a string as place holders for other values. Using string interpolation is a more convenient alternative to string concatenation. For example, `'Hello, %s. Are you going to %s on %s?' % (name, activity, day)` evaluates to the string `'Hello, Albert. Are you going to program on Thursday?'`, if the variables have those corresponding values.

string - A value made up of text. Strings are typed in with a single quote ' or double " on either side. For example, `'Hello'`

substring - A subset of a string value. For example, if `spam` is the string `'Hello'`, then the substring `spam[1:4]` is `'ell'`. Similar to a list slice.

symbols - In cryptography, the individual characters that are encrypted.

syntax error - An error that occurs when the Python interpreter does not understand the source code.

terminate - When a program ends. "Exit" means the same thing.

tracing - To follow through the lines of code in a program in the order that they would execute.

truth tables - Tables showing every possible combination of

tuple - A container data type similar to a list. Tuples are immutable sequence data types, meaning that they cannot have values added or removed from them. For example, `(1, 2, 'cats', 'hello')` is a tuple of four values.

type - see, data types

unordered - In container data types, unordered data types do not have a "first" or "last" value contained inside them, they simply contain values. Dictionaries are the only unordered data type in Python. Lists, tuples, and strings are ordered data types. See also, sequence.

user - The person using the program.

value - A specific instance of a data type. `42` is a value of the integer type. `'Hello'` is a value of the string type.

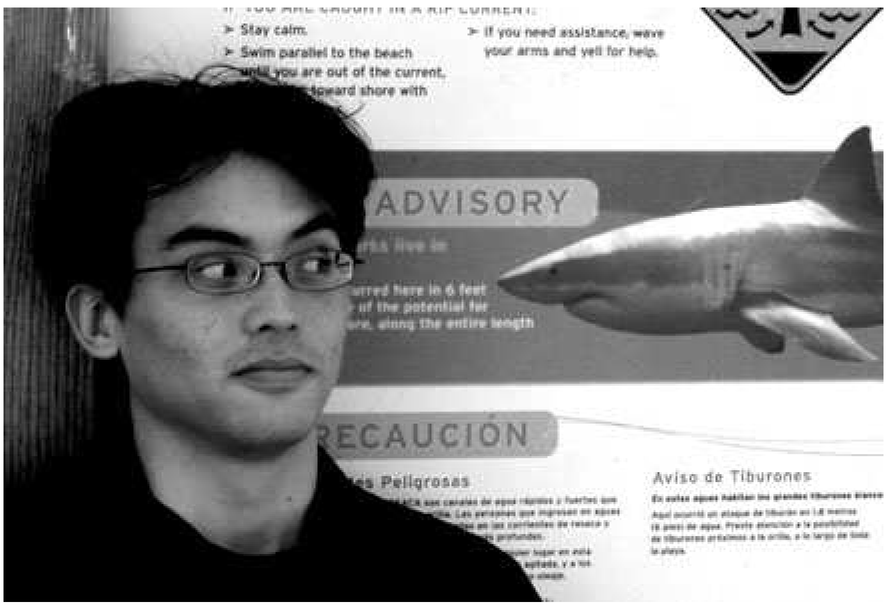
variables - A container that can store a value. List variables contain references to lists.

while loop statement - The `while` keyword, followed by a condition, ending with a : colon character. The `while` statement marks the beginning of a `while` loop.

x-axis - In cartesian coordinate systems, the horizontal (left-right) coordinate axis.

y-axis - In cartesian coordinate systems, the vertical (up-down) coordinate axis.

About the Author



Albert Sweigart (but you can call him Al), is a software developer in San Francisco, California who enjoys bicycling, reading, volunteering, network security, haunting coffee shops, and making useful software.

He is originally from Houston, Texas. He finally put his University of Texas at Austin computer science degree in a frame. He is a friendly introvert, a cat person, and fears that he is losing brain cells over time. He laughs out loud when watching park squirrels, and people think he's a simpleton.

His web site and blog are located at http://coffeeghost.net